Apple II

Apple Pascal

Operating System Reference Manual

NOTICE

Apple Computer Inc. reserves the right to make improvements in the product described in this manual at any time and without notice.

DISCLAIMER OF ALL WARRANTIES AND LIABILITY

APPLE COMPUTER INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL OR WITH RESPECT TO THE SOFTWARE DESCRIBED IN THIS MANUAL, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. APPLE COMPUTER INC. SOFTWARE IS SOLD OR LICENSED "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAMS PROVE DEFECTIVE FOLLOWING THEIR PURCHASE, THE BUYER (AND NOT APPLE COMPUTER INC., ITS DISTRIBUTOR, OR ITS RETAILER) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL APPLE COMPUTER INC. BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE, EVEN IF APPLE COMPUTER INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

This manual is copyrighted. All rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Apple Computer Inc.

©1980 by APPLE COMPUTER INC.
10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

All rights reserved.

The word Apple and the Apple Logo are registered trademarks of APPLE COMPUTER INC.

APPLE Product #A2L0028
(030-0100-00)

Apple II

Apple Pascal

Operating System Reference Manual

ACKNOWLEDGEMENTS

The Apple® Pascal system incorporates UCSD Pascal™ and Apple extensions for graphics and other functions. UCSD Pascal was developed largely by the Institute for Information Science at the University of California at San Diego, under the direction of Kenneth L. Bowles.

"UCSD PASCAL" is a trademark of The Regents of The University of California. Use thereof in conjunction with any goods or services is authorized by specific license only and is an indication that the associated product or service has met quality assurance standards prescribed by the University. Any unauthorized use thereof is contrary to the laws of the State of California.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION 1

2 How to Use This Manual
4 Manual Organization Summary

CHAPTER 2

THE COMMAND LEVEL 5

6 The Operating System
9 Commands Usable at All Levels
11 Using the Command Level
16 The Command Level Options
20 Command Option Summary

CHAPTER 3

THE FILER 22

24 Introduction
26 Volumes
28 Files
33 Using the Filer
34 The Filer Commands
67 Filer Command Summary

CHAPTER 4

THE EDITOR 70

72 Introduction
77 A Brief Scenario
83 A Little More Detail
89 The Editor Commands
124 Editor Command Summary

CHAPTER 5

THE PASCAL COMPILER

127

128 Introduction
130 Using the Compiler

CHAPTER 6

THE 6502 ASSEMBLER

134

136 Introduction
138 Using the Assembler
151 Assembler Information
157 The Assembler Directives
172 Assembler Directive Summary

CHAPTER 7

THE LINKER

175

176 Introduction
178 Using the Linker

CHAPTER 8

UTILITY PROGRAMS

183

185 Introduction
185 Formatting New Diskettes
187 The System Librarian
195 Library Mapping
200 System Reconfiguration
211 Changing GOTOXY Communication
215 Removing Linefeed from Return
217 Calculator
219 Utilities Summary

APPENDIX A

ARCHITECTURE OF THE P-MACHINE 223

224 Technical Information
229 The P-Machine Instruction Set

APPENDIX B

OPERATION OF THE P-MACHINE 247

248 Introduction
248 The System Codefile
254 System Memory Use
260 Overview

APPENDIX C

FILE FORMATS 265

266 Text Files
266 Data Files
266 Code Files

APPENDIX D

TABLES 271

272 When to Use .TEXT and .CODE
273 Language System Diskette Files
276 Pascal I/O Device Volumes
277 Apple I/O Device Slots
278 Execution Error Messages
280 I/O Error Messages
281 6502 Assembler Error Messages
283 ASCII Character Codes
284 P-Machine Op-Codes

APPENDIX E

OPERATING SYSTEM SUMMARY 285

286 Operating System
286 Command Level
287 Filer
288 Editor
289 Compiler
289 Assembler
289 Linker
290 Utilities

INDEX 292

Inside Back Cover Apple Pascal Command Tree

CHAPTER 1

INTRODUCTION

2 HOW TO USE THIS MANUAL
2 Getting Started
2 The Operating System
3 The Language
4 MANUAL ORGANIZATION SUMMARY

HOW TO USE THIS MANUAL

The Apple Pascal system is intended to run on the Apple II and Apple II-Plus computers. The system requires 48K bytes of installed memory, one or more Apple Disk II disk drives and the Apple Language System. Installation procedures and other instructions for the required Language Card are covered in the small Apple Language System manual, which you should read before beginning this manual.

👁

The above symbol appears throughout this manual. Its purpose is to alert you to an unusual feature of the Apple Pascal operating system.

GETTING STARTED

The Apple Pascal Operating System Reference Manual and the language reference manual for the programming language you will use with the Appple Pascal operating system are most definitely not intended for beginners at using computers and Pascal. However, each language reference manual for use with the Apple Pascal operating system includes easily followed chapters to help you begin using the operating system with that programming language. Read these chapters in the programming language reference manual FIRST. They will help introduce you to the Apple Pascal operating system and help you get the "feel" of things, after which the more technical material in the Apple Pascal Operating System Reference Manual will be easier to follow.

THE OPERATING SYSTEM

The Apple Pascal system includes a Filer for handling disk files, a powerful text Editor for writing programs, a Pascal Compiler to convert your programs into executable P-code, a 6502 Assembler to convert assembly-language routines into machine-language code, and a Linker to combine other routines into your program. These make up the Apple Pascal operating system: they are not part of the Pascal programming language itself, but they help you to write, store, and execute your programs. In Chapters 2 through 7 of this manual, you will find detailed discussions of each portion of the operating system. The "command tree" shown on the inside back cover will help you find your way around in the various levels of the operating system.

In addition to the main operating system, there are also various utility programs which let you format new diskettes, put routines into a system library, and configure your system to run with most external terminals. These utility programs and others are discussed in Chapter 8.

In each chapter about the operating system, a special "Diskfiles" section tells you which Language System diskette to put in each disk drive before attempting to use that portion of the operating system. These "Diskfiles" sections can be a great help, especially for one-disk-drive systems, where use of the Language System diskettes may seem confusing, at first.

In general, each chapter in this manual contains a detailed discussion of a particular portion of the Apple Pascal operating system, followed by a summary of the information in that chapter. Read the main body of the chapter the first time, and whenever you need detailed information about that topic. Use the summary for a quick reference, when you just need to be reminded of information you already know.

An even briefer summary of all the operating system commands appears in the last appendix, at the end of this manual.

THE LANGUAGE

This manual contains some information about using the Apple Pascal operating system with the Apple Pascal programming language and with 6502 assembly language. However, it does not attempt to describe or explain the details of any programming language. For further information about any of the programming languages used with the Apple Pascal operating system, you should consult the reference manuals for the individual languages.

MANUAL ORGANIZATION SUMMARY

Chapter 1	INTRODUCTION
Chapter 2	COMMAND level: to select Filing, Editing, Compiling, Assembling, Linking, Running, etc.
Chapter 3	FILER: handles disk and other files
Chapter 4	EDITOR: for writing and changing text files
Chapter 5	COMPILER: converts Pascal text into P-code
Chapter 6	ASSEMBLER: converts assembly-language text into 6502 machine-language
Chapter 7	LINKER: ties external routines into your programs
Chapter 8	UTILITIES: disk formatter, installing routines into a system library, external terminal setup, etc.
Appendix A	Architecture of the P-machine
Appendix B	Operation of the P-machine
Appendix C	File formats
Appendix D	TABLES of useful information
Appendix E	SUMMARY of all operating system commands

CHAPTER 2

THE COMMAND LEVEL

6 THE OPERATING SYSTEM
6 The Screen Display
6 The Prompt Line
7 Diskfiles Needed for Booting
8 Making a Turnkey System
9 The Workfile
9 COMMANDS USABLE AT ALL LEVELS
9 CTRL-A
10 CTRL-Z
10 CTRL-@
10 CTRL-F
10 CTRL-S
10 Power Down-and-Up
10 RESET
11 USING THE COMMAND LEVEL
11 Diskfiles Needed
16 THE COMMAND LEVEL OPTIONS
16 F(ile
16 E(dit
16 C(ompile
17 A(ssemble
17 L(ink
17 X(ecute
18 R(un
19 D(ebug
19 U(ser restart
19 I(nitialize
19 H(alt
20 COMMAND OPTION SUMMARY

THE OPERATING SYSTEM

The Apple Pascal system described in this document is intended to run on the Apple II and Apple II-Plus computers. The system requires 48K bytes of installed memory, one or more Apple Disk II disk drives and the Apple Language System.

While the system is primarily intended to use the Apple keyboard and the usual TV or monitor, an external CRT terminal such as the Soroc IQ 120 can act as the CONSOLE device, connected to the Apple through a modified Apple Communications Interface Card. With such an external terminal, it becomes possible to do text and program editing in upper and lower case on a large (80 characters by 24 lines) screen. For most programming purposes, an external terminal is completely unnecessary.

This manual is written specifically for using the Apple Pascal operating system with the Apple Pascal programming language and compiler. If you are using the Apple Pascal operating system with any other programming language, you must first read that language's reference manual for special instructions about using this operating system.

THE SCREEN DISPLAY

The Apple Pascal operating system always uses a display that is 80 characters wide. The Apple's 40-character screen normally shows only the leftmost 40 characters (the left "page") of the Pascal display, which is sufficient for most applications. To see the rightmost 40 characters (the right "page") of the display, type A while holding down the CTRL key (we will usually call this "CTRL-A"). Press CTRL-A again to go back to the left "page" of the display. When the white square cursor is on the screen, you can make the Apple screen scroll right and left to "follow" the cursor automatically, by pressing CTRL-Z. CTRL-A (like many other commands) cancels CTRL-Z.

THE PROMPT LINE

At most times you will see a "prompt line" which shows the command options available to you at the moment. Here are 2 examples:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG, ?
```

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(IN
```

Note: In general, prompt lines throughout this manual are given in two somewhat different forms. The longer, more complete form is the prompt line as it might appear on an external terminal which can display an 80-character line. The shorter form is the prompt line as it will appear on the Apple's 40-character monitor or TV screen. In a few cases, the order of the options shown in the long form differs from that in the shorter form. Many prompt lines end in a series of letters and numbers enclosed in square brackets. These indicate the version number of the portion of the program with which you are working.

In response to this prompt line, you can use the Editor, Run a program, operate the Filer, or choose any of several other options, just by typing a single letter. Typing E , for example, will invoke the Editor.

When you do invoke the Editor (or exercise almost any of the other options), you will usually be shown another prompt line that allows you to choose command options appropriate to that activity. The "command tree" shown on the inside back cover of this manual will help you find your way around in the various levels of the operating system.

Sometimes the prompt line contains too many options to fit on the screen's two 40-character "pages". If this happens, a question mark (?) may appear at the end of the prompt line. Whether or not the ? appears in the prompt line, typing ? will cause any remaining command options to be displayed.

On some occasions, you will have to type a name or other information longer than a single character. These entries are terminated by pressing the RETURN key. If you make a typing mistake before pressing the RETURN key, you can backspace over the error by pressing the backspace key (left-pointing arrow key) near the right side of the keyboard. Typing a CTRL-X will erase all the characters you have just typed. On a terminal such as the Soroc IQ 120, a RUB key (or another key) may do this quick-erase. If you wish to get rid of the question altogether, just press the RETURN key.

DISKFILES NEEDED FOR BOOTING

The following diskfile is needed for the first stage of a "cold boot" of the system:

SYSTEM.APPLE	(in the boot drive; required)

The system is "cold booted" every time the Apple's power is turned on, or when the H(alt option is selected from the COMMAND prompt line. The file SYSTEM.APPLE contains the interpreter, which allows compiled P-code to be executed by Apple's 6502 processor. The interpreter is loaded into the Language Card's memory, and write-protected there. It does not have to be re-loaded until the next cold boot of the system. This file is normally found on diskette APPLE1: and also on diskette APPLE3:, so either of those diskettes may be placed in the boot drive (volume #4:) to begin a cold boot.

The following diskfiles are needed to complete a "cold boot" of the system, or to effect a "warm boot" of the system:

SYSTEM.PASCAL	(in the boot drive; required)
SYSTEM.MISCINFO	(in the boot drive; required)

The system is "warm booted" when the Apple's RESET key is pressed, when the I(nitialize option is selected from the COMMAND prompt line, or when any system error causes the system to be re-initialized (re-booted). These two files are normally found on diskette APPLE0: and also on diskette APPLE1:, so either of those diskettes may be placed in the boot drive (volume #4:) to effect a warm boot, or to complete a cold boot. The diskette which supplies these two files becomes the system's "boot diskette".

In general then, it is easiest to cold-boot any system with diskette APPLE1: in the boot drive. Once this cold boot is complete, one-drive users may wish to switch to APPLE0: as their system diskette, by placing APPLE0: in the boot drive and pressing the RESET key. A warm boot may be carried out if either diskette APPLE0: or diskette APPLE1: is in the boot drive.

Note: If APPLE3: is used to start a cold boot, no message appears on the screen. When all action ceases, put APPLE0: or APPLE1: in the boot drive and press the RESET key to complete booting.

MAKING A TURNKEY SYSTEM

The Apple Pascal system allows you to set up a turnkey system, which will automatically begin running a particular program when the Apple is turned on. To set up your Apple as a turnkey system, first make a copy of diskette APPLE1: and use the Filer's C(hange command to change the copy's name to something you will recognize. For example, you might name this diskette TURNKEY: . Now T(ransfer a copy of your program codefile onto the turnkey diskette, giving this new copy of your program the filename SYSTEM.STARTUP . Make sure your turnkey diskette contains the following files:

SYSTEM.APPLE	
SYSTEM.PASCAL	
SYSTEM.MISCINFO	
SYSTEM.LIBRARY	(if needed by your STARTUP program)
SYSTEM.CHARSET	(if needed by your STARTUP program)
SYSTEM.STARTUP	

You may remove any other files (such as SYSTEM.FILER, SYSTEM.EDITOR, and SYSTEM.SYNTAX) if you need more space on the diskette for your program's files.

To run your turnkey program, put the turnkey diskette in the boot drive and turn on the Apple's power. Soon, and with no further intervention, SYSTEM.STARTUP is executed. Thereafter, SYSTEM.STARTUP will also be executed each time the system is re-booted, re-initialized, or the RESET key is pressed.

THE WORKFILE

The Apple Pascal system makes frequent use of a "workfile". The workfile is a special "default file" used during the development of a program or a piece of text. You can Edit, Save and Update, Compile or Assemble, Link and Run the workfile as often as you wish, without having to specify the name of the workfile for each operation. These operations automatically assume you are referring to the workfile on the boot diskette, if there is a workfile currently stored on that diskette, and if that diskette can be found in one of the disk drives. The boot diskette is the diskette that was in the boot drive, volume #4: (slot 6, drive 1), the last time the system was booted.

The system always stores the workfile on the boot diskette, using the same filename: SYSTEM.WRK . This is handy for you and the system, as it makes it easy to find the current file on which you are working. For text, the stored workfile's full name is always SYSTEM.WRK.TEXT . For programs, the stored workfile often consists of both the text version (SYSTEM.WRK.TEXT) and the compiled or assembled version (SYSTEM.WRK.CODE) which are saved and retrieved together. Individual commands automatically use the correct version of the workfile.

It is also possible to designate any other filename as the next workfile, using the Filer's G(et command. This command removes any old files named SYSTEM.WRK from the boot diskette, but does NOT create a new file named SYSTEM.WRK . Instead, the next time any command (such as Edit, Compile, or Run) attempts to use the workfile, the file designated by G(et is used.

Only one workfile is allowed at any one time. This is no limitation, since it is easy to Save your current workfile under a filename of your choosing, so that you can create another workfile. And it is just as easy to bring a saved file back to be your new workfile. These operations are covered in this manual's chapter THE FILER.

COMMANDS USABLE AT ALL LEVELS

Certain system commands can be executed at ANY level of the operating system, regardless of which option is in force at the moment. You have already been introduced to those affecting the screen, but there are others, as well. These omni-present commands are listed below; at no time do these commands appear on any prompt line. Note that the system will detect a typed command only when the next input or output operation begins.

CTRL-A

Shows the other 40-character "page" of the Apple Pascal system's 80-character display, until the next CTRL-A.

CTRL-Z

Initiates "Auto-follow" mode: the screen scrolls right and left to follow the cursor. Cancelled by CTRL-A and many other commands.

CTRL- @

Causes current program to be interrupted and issues the message "PROGRAM INTERRUPTED BY USER." Press spacebar to reinitialize the system.

CTRL-F

Causes subsequent program output to be flushed. The program continues to run, but its output is not sent to the screen or the printer. Cancelled by the next CTRL-F .

CTRL-S

Stops any on-going operating system process or program. When the next CTRL-S is typed, the process continues.

POWER DOWN-AND-UP

Turning the Apple's power switch off and then on again does a "cold boot" of the system, just as if the system were being turned on for the first time. This command will stop any on-going process, at the expense of losing whatever is in the Apple's memory. When the system "hangs" (stops and does not respond to the keyboard, even when you press the RESET key), this command will usually re-start the system. After this command, you will have to repeat the entire normal startup procedure. The P-code interpreter is loaded into the Language Card and write-protected there, so the file SYSTEM.APPLE must be on the diskette in the boot drive, volume #4:. To accomplish a cold boot, one-disk-drive systems and multiple-disk-drive systems must both start with diskette APPLE1: in the boot drive.

RESET

Pressing the Apple's RESET key does a "warm boot" of the system. This command will stop almost any ongoing process, at the expense of losing whatever is in the Apple's memory. When the system "hangs" (stops and does not respond to the keyboard), this command will usually re-start the system. The P-code interpreter is not re-loaded into the Language

Card by a warm boot, so the diskette file SYSTEM.APPLE need not be present. To accomplish a warm boot, either diskette APPLE1: or diskette APPLE0: must be in the boot drive, volume #4 .

USING THE COMMAND LEVEL

The Command level of the Apple Pascal system is reached whenever you boot or reset the system (by any means), when the system re-initializes itself after a fatal run-time error, when you Q(uit the Editor or the Filer, and when you finish C(ompiling, A(ssembling, L(inking, X(ecuting, or R(unning any utility or other program. You have already seen the COMMAND prompt line:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG, ?
```

Use CTRL-A to see the rest of the prompt line. After typing a ? , the remaining Command options are shown:

```
COMMAND: U(SER RESTART, I(NITIALIZE, H(ALT
```

DISKFILES NEEDED

The Apple Pascal operating system is much too large a program to be kept entirely in the Apple's memory all the time. Besides, you use only a small part of the operating system at any given time. For this reason, the operating system is broken into several smaller portions, and these program portions are stored in separate diskfiles on the system diskettes, under filenames such as SYSTEM.FILER, SYSTEM.EDITOR, and SYSTEM.COMPILER. Each option from the COMMAND prompt line uses one or more of these special diskfiles.

Before you specify a particular Command option, you must first make sure that the diskfiles needed by that portion of the operating system are available. In most cases, the required diskfile is allowed to be on ANY diskette in any of your system's disk drives. The system just goes through the diskettes in every drive until it finds a file with the necessary filename.

The workfile (SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE) and the Compiler's error-message file (SYSTEM.SYNTAX) will be found by the system ONLY if they are on the boot diskette. The boot diskette is the diskette that was in the boot drive, volume #4: (slot 6, drive 1), the last time the system was booted. The system will usually look for the boot diskette in the boot drive first, but will also look in the other drives if necessary.

The diskfiles SYSTEM.PASCAL and SYSTEM.LIBRARY are usually accessed by going directly to their positions on the diskette in the boot drive. During the booting process, the system notes the positions of these two files on the boot diskette. Thereafter, whenever the system needs either of those files, it assumes that the diskette in the boot drive still contains the needed files in exactly the same diskette positions occupied by those files during the last boot. If the system does not find the correct information at those diskette locations, the system may "hang" (stop responding to the keyboard), and you will have to re-boot. If you change the contents or the position of either of these files, you should RESET or I(nitialize the system to let it discover the change.

During program execution, the information in SYSTEM.LIBRARY is accessed by its boot-time position in the boot drive. During linking, however, the Linker searches for *SYSTEM.LIBRARY by name, and will find it on the boot diskette in any drive.

For information about entering the Command level via a "cold boot" or via a "warm boot", see the section DISKFILES NEEDED FOR BOOTING, earlier in this chapter. The system is "cold booted" when the Apple's power is turned on, or when the Command option H(alt is selected. The system is "warm booted" when the Apple's RESET key is pressed, when the Command option I(nitialize is selected, or when any system error causes the system to be re-initialized (re-booted).

The following diskfile is needed each time the system returns to the Command level following the termination of any option or program:

SYSTEM.PASCAL	(boot diskette, in boot drive; required)

This file contains the Command level portion of the operating system. The system tries to re-enter the Command level after any program is terminated by Q(uitting any portion of the operating system, by reaching the END. of any option or any program that you are executing, or by any non-fatal execution error.

When the system attempts to return to Command level it first checks to be sure the diskette in the boot drive is the correct boot diskette. If the diskette is in the boot drive, but does not have SYSTEM.PASCAL in the expected diskette locations, the system may "hang", and may not respond even to the RESET key. In this case, you will have to cold boot the system by turning the power off and then on again, with diskette APPLE1: in the boot drive.

Each return to Command level must find the boot diskette in the boot drive. The easiest way to accomplish this is to make sure your boot diskette is in the boot drive whenever a Command option or program is terminated. If a return to Command level finds the wrong diskette in the boot drive, you will be prompted

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette). The boot drive will start again and again, until you press RESET or put the correct boot diskette in the boot drive.

The file SYSTEM.PASCAL is normally found on diskette APPLE0: and also on diskette APPLE1: . However, you must stick with one or the other (the one which is your boot diskette) when leaving the Command level and returning to it. To change from using one of these diskettes to using the other as your boot diskette, you should place the new diskette in the boot drive and press the Apple's RESET key to re-boot. You should also re-boot the system any time you move the file SYSTEM.PASCAL on your system diskette. Re-booting lets the system discover a new boot-diskette's name, and also discover SYSTEM.PASCAL's new diskette location.

The following table summarizes the diskfiles needed by various command options:

COMMAND	FILES NEEDED	WHERE FILES MUST BE FOUND
F(ile	SYSTEM.FILER	(any disk, any drive; needed only at start)
	Files to be moved	(any disks, any drives; T(ransfer requires source file to be present; can prompt for destination file)
E(dit	SYSTEM.EDITOR	(any disk, any drive)
	Textfile to be Edited	(any disk, any drive; optional; default boot disk's SYSTEM.WRK.TEXT, any drive)
C(ompile	SYSTEM.COMPILER	(any disk, any drive)
	Textfile to be Compiled	(any disk, any drive; default is boot disk's SYSTEM.WRK.TEXT, any drive)
	SYSTEM.LIBRARY	(boot disk, boot drive; required only if program USES Intrinsic Units)
	SYSTEM.EDITOR	(any disk, any drive; optional; to fix errors found by Compiler)
	SYSTEM.SYNTAX	(boot disk, any drive; optional; provides error messages on entering Editor)
A(ssemble	SYSTEM.ASSMBLER	(any disk, any drive)
	6500.OPCODES	(any disk, any drive; required)
	6500.ERRORS	(any disk, any drive; optional; provides error messages in Assembler)
	Textfile to be Assembled	(any disk, any drive; default is boot disk's SYSTEM.WRK.TEXT, any drive)
	SYSTEM.EDITOR	(any disk, any drive; optional; to fix errors found by Assembler)
L(ink	SYSTEM.LINKER	(any disk, any drive; needed only to start)
	Host codefile	(any disk, any drive; default is boot disk's SYSTEM.WRK.CODE, any drive)
	Library codefile	(any disk, any drive; default is boot disk's SYSTEM.LIBRARY in any drive)

X(ecute	Codefile to be eXecuted	(any disk, any drive; required only when loading, if no segment overlays)
	SYSTEM.LIBRARY	(boot disk, boot drive; required if the program uses long integers, does file I/O using reals or SEEK, or USES Intrinsic Units)
	SYSTEM.CHARSET	(any disk, any drive; required if WCHAR or WSTRING called from TURTLEGRAPHICS)
R(un	Text or Codefile to be Run	(any disk, any drive; default is boot disk′s SYSTEM.WRK.CODE or .TEXT)
	SYSTEM.COMPILER	(any disk, any drive; required only if file being Run is a textfile)
	SYSTEM.EDITOR	(any disk, any drive; optional; to fix errors found by Compiler)
	SYSTEM.SYNTAX	(boot disk, any drive; optional; provides error messages on entering Editor)
	SYSTEM.LINKER	(any disk, any drive; required only if other routines need to be Linked in) (no Link needed to USE Intrinsic Units)
	SYSTEM.LIBRARY	(boot disk, any drive; required to hold needed routines if Linker called)
	SYSTEM.LIBRARY	(boot disk, boot drive; required if program uses long integers, does file I/O using reals or SEEK, or USES Intrinsic Units)
	SYSTEM.PASCAL	(boot disk, boot drive; required between Compiling, Linking, and eXecuting.
	SYSTEM.CHARSET	(any disk, any drive; required only if program uses WCHAR or WSTRING from TURTLEGRAPHICS)
U(ser restart	All files needed by last program or option	(same file locations required by last program or option)
I(nitialize	SYSTEM.PASCAL SYSTEM.MISCINFO	(any disk, boot drive; this disk becomes the new boot disk)
H(alt	SYSTEM.APPLE	(any disk, boot drive; first stage boot)
	SYSTEM.PASCAL SYSTEM.MISCINFO	(any disk, boot drive; this disk becomes the new boot disk)
Return to Command Level	SYSTEM.PASCAL	(boot disk, boot drive)

The "boot drive" is volume #4: (slot 6, drive 1). The "boot disk" is the diskette that was in the boot drive the last time the system was booted (usually APPLE0: on one-drive systems, and APPLE1: on larger systems).

Here is where the system files needed by the Command level are normally found:

Diskette APPLE0:	Diskette APPLE1:	Diskette APPLE2:
	SYSTEM.APPLE	SYSTEM.COMPILER
SYSTEM.PASCAL	SYSTEM.PASCAL	
SYSTEM.MISCINFO	SYSTEM.MISCINFO	SYSTEM.LINKER
SYSTEM.COMPILER		SYSTEM.ASSMBLER
SYSTEM.EDITOR	SYSTEM.EDITOR	6500.OPCODES
SYSTEM.FILER	SYSTEM FILER	6500.ERRORS
SYSTEM.LIBRARY	SYSTEM.LIBRARY	
SYSTEM.CHARSET	SYSTEM.CHARSET	
SYSTEM.SYNTAX	SYSTEM.SYNTAX	

As you can see, there is little difference between diskettes APPLE0: and APPLE1: . Diskette APPLE0: is more convenient for editing and running Pascal programs, especially on one-drive systems, because it contains the file SYSTEM.COMPILER . However, APPLE0: cannot be used to cold-boot the system, as it is lacking the file SYSTEM.APPLE . Diskette APPLE1: contains all the files you need for editing text, and for cold-booting the system. However, APPLE1: cannot be used to R(un or C(ompile your text, as it is lacking the file SYSTEM.COMPILER . On multiple-drive systems, APPLE1: is often kept in the boot drive and APPLE2: in another drive, thus making all of the operating system available at all times.

Most system files must be available in one of the disk drives constantly, from the moment you select the Command option using that file until you Quit that option or until it terminates. This is true of the E(dit, C(ompile, and A(ssemble options. These programs are "overlaid", using segment procedures, so that different portions of the program are called in from disk as they are needed, to conserve memory. In such cases, if your system has only one disk drive you should use the Filer to T(ransfer all the necessary files to one diskette (usually the system diskette, APPLE0: or APPLE1:) before you select that option.

Files containing non-overlayed programs are needed only at the moment the program is loaded into the Apple's memory. Once they have begun execution, the source diskette may be removed from its drive.

The F(ile Command and L(ink Command options have been purposely written without overlaying. The file SYSTEM.FILER is needed at the moment you select the F(ile option, but not subsequently. You should make sure a diskette containing SYSTEM.FILER is in one of the disk drives when you select the F(ile option, but as soon as the new FILER prompt line appears you may remove that diskette and put in any other.

Any time the Linker is invoked, SYSTEM.LINKER must be available. However, when the LINKER prompt line appears, SYSTEM.LINKER is no longer necessary and the diskette containing SYSTEM.LINKER may be removed from the system to make room for other diskettes.

More details about using the disks with various commands are given in the chapters on the Filer, Editor, Compiler, Assembler, and Linker.

THE COMMAND LEVEL OPTIONS

Many of the Command options are explained only briefly below. This manual's chapters on the Filer, the Editor, the Compiler, the Assembler, and the Linker discuss individual Command options in much greater detail.

F(ILE

Typing F from Command level places you in a level of the system called the Filer. The Filer contains commands for saving, reading, moving and deleting the workfile and other disk files. Other commands tell you what peripheral devices and diskettes are currently available to the system, and what files are saved on each diskette. Still other commands let you check diskettes for damage or recording errors, and let you set the system's current date and the default volume name. For more documentation, see this manual's chapter THE FILER.

E(DIT

Typing E while at the Command level invokes the Editor program. If a workfile is available, that file is automatically read into the computer for editing. Otherwise, the Editor asks you to specify a textfile or begin creating a new one. While in the Editor, you may create or alter text in the workfile or in any textfile. Various commands allow you to insert and delete information, find and replace specified character strings, change the text format, combine files, etc. On leaving the Editor, you may save your edited text in the updated workfile or in another specified file. See this manual's chapter THE EDITOR for details.

C(OMPILE

Typing C while at the Command level invokes the system Compiler. If a text workfile is available, that file is automatically read into the computer for compiling. Otherwise, the Compiler asks you to specify a source textfile and an object codefile. During compilation, if the Compiler detects a syntax error, it gives you the option of calling the Editor, which points out the error and lets you correct it. After a successful compilation, the resulting P-code is saved in the code workfile unless you have previously specified another object codefile. For more details, see this manual's chapter THE PASCAL COMPILER.

A(SSEMBLE

Typing A from the Command level invokes the 6502 Assembler program. If a text workfile is available, that file is automatically read into the computer for assembling. Otherwise, the Assembler asks you to specify a source textfile and an object codefile. During assembly, if the Assembler detects a syntax error, it gives you the option of calling the Editor, which points out the error and lets you correct it. After a successful assembly, the resulting machine code is saved in the code workfile unless you have previously specified another object codefile. For more information, see this manual's chapter THE 6502 ASSEMBLER.

L(INK

Typing L from the Command level starts the system Linker program explicitly. Unlike the automatic linking initiated by the R(un command (see description below), this option allows you to link previously compiled or assembled routines into your program, taking those routines from SYSTEM.LIBRARY or any other specified library file. For more information, see this manual's chapter THE LINKER.

X(ECUTE

After typing X from the Command level, the system asks you to specify a previously compiled codefile:

```
EXECUTE WHAT FILE?
```

You should respond by typing the filename of the compiled P-code program that you wish to be executed.

It is not necessary to type the suffix .CODE ; that suffix is automatically supplied by the system if you don't type it. If you wish to defeat this feature, in order to execute a program whose filename does not have a .CODE suffix, type a period (.) after the last character of the desired filename.

When you have specified a codefile, that file is executed if it is available, except in the following cases:

1) If all code necessary to execute the Pascal codefile has not yet been linked in, or if the file is an unlinked assembly codefile, the message

```
MUST L(INK FIRST
```

is displayed (see L(ink, above). You are immediately returned to the Command level, where you may carry out the necessary linking process before executing the linked file.

2) If the specified file contains anything other than the expected compiled P-code, (for example, text or linked assembly code) you will get a message similar to this:

```
MYDISK MYFILE.CODE NOT CODE
```

3) If the file SYSTEM.LIBRARY is not available in the expected diskette locations in the boot drive (#4:), or if that file is not complete, you may be shown the message

```
REQUIRED INTRINSIC(S) NOT AVAILABLE
```

This indicates that the program you are executing uses Units or routines normally found in the file SYSTEM.LIBRARY . These include routines for long integers, random numbers, transcendental functions, game paddles, graphics, and file input and output using real numbers or SEEK.

4) If the file SYSTEM.CHARSET is not available on the boot diskette, a program using WCHAR or WSTRING from the Unit TURTLEGRAPHICS will be executed, but no characters will appear on the screen.

It is convenient to X(ecute programs which have already been compiled, but which are not currently in the workfile. Otherwise, you would have to enter the Filer, G(et the file (this identifies it as the next workfile), Q(uit the Filer, and then R(un the program.

The most common operating system functions can be selected directly from the prompt lines. Functions used more rarely are supplied as utility programs, which are available through the X(ecute command. By X(ecuting these utilities, you can format new diskettes, place compiled or assembled routines in the system library, configure your system to run with an external terminal, etc. For a complete discussion of these abilities, see this manual's chapter UTILITY PROGRAMS.

R(UN

Typing R from the Command level initiates the R(un sequence, which combines the Command options C(ompile, L(ink, and X(ecute, as needed. If a code workfile is available, that file is automatically executed. Otherwise, the Compiler is automatically called as described above. If the compilation requires linkage to other routines, the Linker is automatically invoked and looks for the routines only in the file SYSTEM.LIBRARY on the boot diskette. After successful compilation and linking (if those were necessary), the program is executed. See the descriptions of the options C(ompile, L(ink, and X(ecute for more details.

Note: Between any two portions of the R(un sequence, the system returns for an instant to the Command level. Thus the boot diskette must normally remain in the boot drive throughout the R(un sequence.

The R(un option is one part of the normal program-development process. Initially, the E(dit option is used to create the first executable portion of a new program. When this portion seems complete, it is saved in the workfile. Then, the R(un option attempts to compile the workfile and, if successful, to execute the compiled program. To expand the program or to correct compilation errors, you go back to the E(dit option. Then the workfile is U(pdated to contain this new version, and you can R(un the workfile again. And so on.

D(EBUG

The D(ebug prompt does not correspond to a currently implemented command. The Compiler is called, but no debugging routines are available. Do not use this command.

U(SER RESTART

Typing U from the Command level tells the system to begin executing again the last program or option which was executed. For example, if you have just Quit the Editor the U(ser restart option will re-invoke the Editor; if you have just finished executing a Pascal program, that program will be executed again.

I(NITIALIZE

Typing I from the Command level has nearly the same effect as pressing the Apple's RESET key. The system is re-started, and everything in memory is lost, but the P-code interpreter is not re-loaded into the Language Card, so the diskette file SYSTEM.APPLE need not be present. Either diskette APPLE1: or diskette APPLE0: must be in the boot drive, volume #4: . I(nitializing is a little faster than the "warm boot" caused by RESET and the P(refix diskette name is not forgotten as it is when RESET is pressed.

H(ALT

Typing H from the Command level does a "cold boot" of the system, just as if the system were being turned on for the first time. The P-code interpreter is loaded into the Language Card and write-protected there, so the file SYSTEM.APPLE must be on the diskette in the boot drive, volume #4: . To accomplish this, one-disk-drive systems and multiple-disk-drive systems must both have the diskette APPLE1: in the boot drive. After the H(alt command, you will have to repeat the entire normal startup procedure.

COMMAND OPTION SUMMARY

Many of these options use the "workfile". The text portion of the workfile is SYSTEM.WRK.TEXT, created on the boot diskette by the Editor's U(pdate command. The code portion of the workfile is SYSTEM.WRK.CODE, created on the boot diskette by the R(un or C(ompile options. In addition, the Filer's G(et command can be used to designate any other text and/or code file as the workfile for the next option to use.

F(ile	Invokes the Filer, which is used to save, move, and retrieve information stored on diskettes.
E(dit	Invokes the Editor, which is used to create and modify text. Reads the workfile or other specified textfile into the Apple for editing.
C(ompile	Invokes the Pascal Compiler, which converts the text of a Pascal program (found in the workfile or other specified textfile) into executable P-code.
A(ssemble	Invokes the Assembler, which converts the text of an assembly-language subroutine (found in the workfile or other specified textfile) into 6502 machine language.
L(ink	Combines external P-code and machine-language subroutines (found in SYSTEM.LIBRARY or other specified library codefile) into a Pascal host program (found in the code workfile or other specified host codefile).
X(ecute	Loads and runs the specified Pascal program codefile.
R(un	Executes the current workfile, automatically compiling and linking (from SYSTEM.LIBRARY) first, if necessary.
D(ebug	Not implemented; do not use this option.
U(ser restart	Attempts to execute again the last program or option that was executed.
I(nitialize	Does a "warm boot" of the system, similar to pressing the RESET key, but faster.
H(alt	Does a "cold boot" of the system, like turning the Apple's power off and then on again.

CHAPTER 3

THE FILER

24 INTRODUCTION
24 Diskfiles Needed
25 Technical Information
26 VOLUMES
26 Input and Output Devices
26 Specifying a Volume
27 Shorthand Volume Names
28 FILES
28 Diskette File Types
29 The Workfile
29 Specifying a File
30   Filenames
30   File Size Specification
30   Shorthand Filename
30   Wildcards
33 USING THE FILER
34 THE FILER COMMANDS
34 General File-Moving Command
34   T(ransfer
42   Copying a Diskette
44 General Diskfile Commands
44   M(ake
45   C(hange
48   R(emove
49   K(runch
50   Z(ero
51 Workfile Commands
51   G(et
53   S(ave
55   N(ew
55   W(hat
55 Information Commands
55   V(olumes
56   L(ist Directory
59   E(xtended Directory List
61 Disk Upkeep Commands
61   B(ad Blocks
62   X(amine
65 Miscellaneous Commands
65   P(refix
65   D(ate
66   Q(uit

67 FILER COMMAND SUMMARY
67 File Specification
67 System Commands
68 General File-Moving Command
68 General Diskfile Commands
68 Workfile Commands
69 Information Commands
69 Disk Upkeep Commands
69 Miscellaneous Commands

INTRODUCTION

The Filer portion of the Apple Pascal operating system handles most of the tasks of transferring information from one place to another. Saving information on disk, moving and deleting disk files, sending information to the computer or to the printer -- these are some of the functions of the Apple Pascal Filer. The Filer is also responsible for telling you where files have been placed on the diskettes, and what devices and diskettes are available for your system's use.

DISKFILES NEEDED

The following diskfile is needed when you type F to select the Filer, from the Command level:

SYSTEM.FILER (any diskette, any drive; required)

When the FILER prompt line appears, SYSTEM.FILER is no longer necessary, and the diskette containing SYSTEM.FILER may be removed from the system to make room for other diskettes. The file SYSTEM.FILER is normally found on diskette APPLE0: and also on diskette APPLE1: , so one of those should be in any available disk drive when you type F from the Command level.

When you use the Filer's T(ransfer command to transfer information from one diskette to another, the source diskette for the transfer must be available in any disk drive before you answer the question TRANSFER? When you have specified the destination for the transfer, you will be prompted to insert the destination diskette if it is not already in a disk drive. This is not a problem with systems that have two or more disk drives, as both source and destination diskettes can be placed in available drives at the same time.

The following diskfile is needed when you type Q to Quit the Filer:

SYSTEM.PASCAL (on boot diskette, in boot drive; required)

This file must be on the boot diskette, in the boot drive, and must occupy the same diskette locations that it occupied when the system was last booted. This means you should place APPLE0: or APPLE1: (whichever became your boot diskette when you last booted the system) back into drive #4: before you type Q to Quit the Filer. If you forget to do this, the system will tell you to

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette). The boot drive will start again and again, until you press RESET or put the correct boot diskette in the boot drive.

TECHNICAL INFORMATION

The Apple Pascal operating system stores information on a diskette in 35 concentric zones or bands, called "tracks". The disk drive's recording and reading head can be moved in and out, to stop and hover over each of these 35 different zones of the spinning diskette.

The length of each track on the diskette is divided into 16 segments, called "sectors". Once the disk drive's recording and reading head is positioned over a given track, that track's 16 sectors will pass under the head, one after the other, each time the diskette spins around.

Each sector consists of an "address field" and a "data field". The address field tells the system exactly which sector of which track is about to be read from or written onto by the disk's read/record head. The address fields are written on a diskette just once, when the diskette is formatted for first use. The data field is the portion of each sector used for storing your data, code, or text information. Up to 256 bytes of information can be stored in each sector's data field.

The Apple Pascal system always stores information in two-sector units called "blocks", each containing 512 bytes (also called "1/2 K" bytes) of information. Each of a diskette's 35 tracks can thus store eight blocks of information, for a total diskette storage capacity of 280 blocks (140 K bytes). While the Filer handles all of this for you automatically, the lowest-level Pascal routines for storing and retrieving diskette information are also available, through the system intrinsics UNITWRITE and UNITREAD (see the Apple Pascal Language Reference Manual for details).

The 280 blocks on a diskette are not all available for storing your programs or other files. Blocks 0 and 1 are reserved for the bootstrap program. In addition, every diskette must contain a "directory", which is the system's only way to recover the other information stored on that diskette. The directory occupies blocks 2 through 5 on the diskette and may store information for up to 77 different files.

A file is stored on the diskette only in contiguous blocks of contiguous tracks. Free blocks which are scattered here and there on the diskette may not be usable to store a large file, but you can use the K(runch command to combine scattered free blocks into one contiguous area that can be used.

If the W(rite or U(pdate Editor option is chosen to save a file, the new version is saved; THEN the old version is deleted. This uses more space on the diskette, but also insures that at all times during the saving process at least one version of your file is intact on the diskette. When the Editor's S(ave option is chosen, the original file is destroyed while the new file is being created. This command asks your permission before it overwrites the old file with the new one.

VOLUMES

INPUT AND OUTPUT DEVICES

A "volume" is any input or output device, such as the screen, the keyboard, or a disk. A "block-structured" device is one that can have a directory and files. In the Apple Pascal sytem, the only block-structured devices are the Disk II floppy disk drives. A non-block-structured device does not have any internal structure; it simply produces or consumes a stream of data. The screen and the keyboard, for example, are non-block-structured.

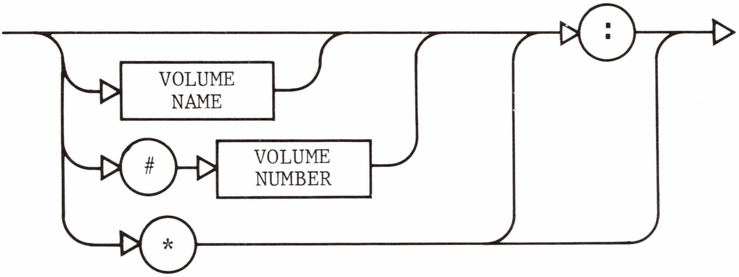
A device may be referred to by its volume number or by its volume name. The volume name of a disk drive is the name of the diskette currently in that disk drive. The following table shows the reserved volume numbers and volume names that Apple Pascal uses to refer to various input and output devices.

Volume Number	Volume Name	Description of Input/Output Device
#0:		(not used)
#1:	CONSOLE:	Screen and keyboard with echo
#2:	SYSTERM:	Reads keyboard without echoing it
#3:		(not used)
#4:	<diskette name>:	Boot disk drive (slot 6, drive 1)
#5:	<diskette name>:	2nd disk drive (slot 6, drive 2)
#6:	PRINTER:	Printer (card in slot 1)
#7:	REMIN:	Remote input (card in slot 2)
#8:	REMOUT:	Remote output
#9:	<diskette name>:	5th disk drive (slot 4, drive 1)
#10:	<diskette name>:	6th disk drive (slot 4, drive 2)
#11:	<diskette name>:	3rd disk drive (slot 5, drive 1)
#12:	<diskette name>:	4th disk drive (slot 5, drive 2)

VOLUME NAMES AND NUMBERS

SPECIFYING A VOLUME

Many Apple Pascal operating system commands require you to specify at least one volume. A complete volume specification consists of the volume name or the volume number for the desired device, followed by a colon (:). The colon is very important: it tells the system that the name or number preceding the colon is a volume specification, and not a diskfile's filename. A stand-alone diskette volume name or number (not followed by a filename) tells the Filer that it is to act in some appropriate way on the diskette AS A WHOLE, and not merely on a certain file on that diskette. If a volume number is specified that is not followed by a filename, the colon following the volume number is optional. The following diagram defines the syntax of volume specification:



VOLUME-SPECIFICATION SYNTAX

The table of VOLUME NAMES AND NUMBERS, given earlier, shows the volume numbers used to specify various input and output devices, and the reserved volume names used to specify non-block-structured devices such as a printer. The volume name for a block-structured volume (a disk drive) is the name that you have assigned to the diskette in that drive. If you specify the volume number of a disk drive, the Filer automatically converts that specification to the volume name of the diskette found in that drive. A diskette's volume name must be seven or fewer characters long and may not contain an equals sign (=), dollar sign ($), question mark (?) or comma (,).

Note: Never issue operating system commands when two diskettes with the same volume name are in the system. Even if you specify the correct drives by their volume numbers, the system will often operate on the wrong diskette (usually the diskette in the higher-numbered drive). If the operation involves updating the diskette's directory, the system may store the wrong diskette's directory onto your diskette, making the files originally on that diskette unavailable. The same problem may occur if you replace the diskette in a drive with another diskette with the same volume name.

Moral: Make SURE the diskettes you use have DIFFERENT volume names.

SHORTHAND VOLUME NAMES

An asterisk (*) can be used to specify the volume name of the "System" or "Boot" diskette, the diskette which was in the boot drive (volume #4:) when the Apple Pascal system was last booted. The Filer's V(olumes command reports the asterisk default volume name as the "ROOT VOLUME".

If a filename is specified with no preceding volume name or number, or if a volume is specified with only a colon (:), the Apple Pascal system supplies the volume name of the "Prefix" volume. Booting the system sets the Prefix to the name of the boot diskette. Thereafter, the Prefix default volume name can be changed at any time by using the Filer's P(refix command. Usually, you will set this to the volume name of the diskette with which you are currently working, to save much typing. However, the Prefix can also be set to other devices, such as the PRINTER: .

FILES

DISKETTE FILE TYPES

A "file" is a collection of information which is stored on the diskette and which may be referred to by a filename. Each diskette has a "directory" which contains the filenames and locations of each file on the diskette. The File handler, or Filer, uses the information contained in the diskette directory to manipulate files.

The use of a file is determined by the file's "type", which specifies the kind of information stored in the file. You and the computer can both tell the file's type when the file is created by looking at a portion of the filename called the "suffix". Here are the filename suffixes normally recognized by the computer, and the associated file types as shown in the rightmost column of an E(xtended directory listing:

Suffix	File Type	E(xtended directory listing
.TEXT	Human-readable text	TEXTFILE
.CODE	Machine-executable code	CODEFILE
.DATA	Data	DATAFILE
.BAD	When created by E(xamine command, an immmovable file covering a physically damaged area of a diskette	BAD FILE
.INFO	(not used)	INFOFILE
.GRAF	(not used)	GRAFFILE
.FOTO	(not used)	FOTOFILE

For instance, a file named MYFILE.TEXT would be treated by the computer as a file containing human-readable text. For more information concerning the internal format of the information contained in different types of files, see Appendix C.

Under some circumstances (after C(hanging the file´s name, for example) the file´s actual type may not agree with it´s filename suffix. The actual type of the file may be determined by examining the rightmost column of the E(xtended directory listing for the file.

THE WORKFILE

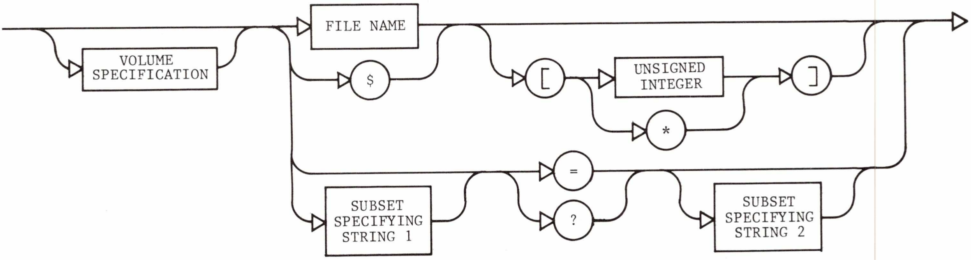
The workfile is a scratch-pad diskette copy of the file currently being worked with. It is automatically taken as the default file by the Command-level options R(un, E(dit, C(ompile, A(ssemble, and L(ink. The workfile can be a single diskette file, or it can consist of two or more diskette files with the same basic filename but different suffixes. Typically, the version with the .CODE suffix is the compiled or assembled version of the file with the .TEXT suffix. Operating system commands for dealing with the workfile automatically choose the correct version of the file.

When a workfile is Edited and Updated, the system stores the new text version on the boot diskette, under the filename SYSTEM.WRK.TEXT . When this new text workfile is compiled or assembled, the resulting code version of the workfile is stored on the boot diskette under the filename SYSTEM.WRK.CODE . If you create (by Edit and Update) a new text version SYSTEM.WRK.TEXT , the old code version SYSTEM.WRK.CODE is automatically erased . You may then create a new code version corresponding to the new text version, by compiling or assembling the workfile.

From the Filer, you can erase the workfile (you must do this before you can Edit, Run, Compile, or Assemble any new file), save the workfile, or designate the next file to be the workfile. These Filer commands automatically operate on all available versions of the workfile.

SPECIFYING A FILE

Many Apple Pascal operating system commands require you to specify at least one file. A complete file specification consists of the volume name or number for the desired device, followed by a colon, followed -- if you are specifying a disk file -- by the filename of the particular disk file desired. This diagram defines file specification syntax:



Filenames

Once a disk device is specified, by its volume number or by the name of the diskette in that drive, any file on that diskette may be specified by its filename. A legal diskette filename can consist of up to 15 characters. In order for the file to be Run, the last five characters must be .TEXT or .CODE . Without a .TEXT or a .CODE suffix, the file may be executed but it may not be put in the workfile. Lower-case letters typed into a filename are translated to upper-case, and spaces and non-printing characters are removed from the filename. All characters are legal in filenames. However, from the keyboard you should not type filenames that include the following characters: dollar sign ($), left square bracket ([), equals sign (=), question mark (?), RETURN, and the CTRL characters C, F, M, S, U, and @.

WARNING: The Filer will not be able to access filenames containing the characters dollar sign ($), equals sign (=), question mark (?), or comma (,).

File Size Specification

It is sometimes possible to specify the size of a disk file immediately after the filename, by enclosing in square brackets [like this] the number of diskette blocks the file is to occupy. There are also two shorthand file size specifications: [0] says the file is to occupy all of the largest unused area, while [*] says the file is to occupy all of the second-largest area or half of the largest area, whichever is larger. If no file size specification is given, the usual default is [0]. The file size specification is primarily useful in the Filer commands T(ransfer and M(ake. However, it can also be used in the Assembler when specifying the output codefile, where [*] is the file-size default.

Shorthand Filename

The T(ransfer command will accept the dollar sign ($) as its SECOND specified filename. This means that the transferred copy of the file is to have the same filename as the original file.

Wildcards

The wildcard characters, equals sign (=) and question mark (?), are used to specify a subset of the filenames on a diskette, by indicating the portion of a filename which may be ignored or which remains unchanged. The Filer performs the requested action on all files whose filenames meet the subset specification. The form of a wildcard filename specification is as follows:

```
        <string1>=<string2>        or        <string1>?<string2>
```

where <string1> and <string2> are sometimes called the "subset-specifying strings". The subset-specifying strings indicate the portion of a filename which may NOT be ignored or which is to be changed. For example, the filename subset specification

```
MYDISK:DOC=TEXT
```

tells the Filer to perform the requested action on all of MYDISK:'s files whose filenames begin with the string DOC and end with the string TEXT . If a question mark is used instead of an equals sign

```
MYDISK:DOC?TEXT
```

the effect is identical except that the Filer requests verification before affecting each file in the specified subset. Instead of a "Y" or an "N" response, you may press the ESC key. This will return you to the outer level of the Filer.

Only one wildcard may be used in a filename specification:

```
MYDISK:DO?TE?T       or       MYDISK:=TE=
```

are NOT legal specifications, because one of the subset-specifying strings contains the forbidden filename character ? or = . An attempt to use the first specification shown above will cause the message

```
TE?T SCAN STRING - ILLEGAL FORMAT
```

The Filer commands T(ransfer and C(hange both require two file specifications. If the first specification contains a wildcard (and the second specification is for a disk volume), the second specification must also contain a wildcard. If you forget, you will be given the message

```
BAD FORM (WILD <TO> NON-WILD) CARD
```

The only legal exception to this rule occurs when T(ransfer is given the $ as the second filename specification.

Either or both of the subset-specifying strings may be empty. For example, a filename subset specification such as =TEXT or DOC= or even just = is valid. This last case, where both subset-specifying strings are empty, is interpreted by the Filer to specify every filename, so typing = or ? alone causes the Filer to perform the appropriate action on every file in the specified diskette's directory. This feature can sometimes be used to act on a file whose filename is not "recognized" by Filer commands (because of illegal characters in the filename, or a slightly damaged directory, say).

The subset-specifying strings may not "overlap". For example, the filename subset specified by GOON=NS would not include the filename GOONS, whereas GOON=S would be a valid (although pointless) specification for the filename GOONS .

EXAMPLE:

Suppose you are given this directory for the diskette volume named MYDISK:

```
NAUGHTYBITS       6   23-JUN-76
MOLD.TEXT         4   29-JAN-57
USELESS.CODE     10   12-MAY-78
MOLD.CODE         4   29-JAN-57
NEVERMORE.TEXT   12    5-APR-74
GOONS             5   10-SEP-52
```

After typing R for R(emove, you will see this message:

```
REMOVE WHAT FILE ?    or    REMOVE ?
```

Response 1: MYDISK:N=

Typing the above response generates the message:

```
MYDISK:NAUGHTYBITS          --> REMOVED
MYDISK:NEVERMORE.TEXT       --> REMOVED
UPDATE DIRECTORY ?
```

At this point you can type Y to remove all the files marked REMOVED, or you can type N , in which case the files will not be removed. The Filer always requests verification before completing any wildcard removes.

Response 2: MYDISK:N?

Typing this response generates the message:

```
REMOVE NAUGHTYBITS ?
```

After you type a response (Y or N), the Filer asks:

```
REMOVE NEVERMORE.TEXT ?
```

Again you may type a response (Y or N), and if you have given any Y responses, the Filer asks:

```
UPDATE DIRECTORY ?
```

As with the previous pattern, this gives you one last chance to change your mind before the files are finally removed.

EXAMPLE:

Again, suppose you have a diskette MYDISK: with the same directory as in the previous example. After typing L for L(dir (which means List the diskette's Directory), you will see this message:

```
DIR LISTING OF WHAT VOL ?   or   DIR LISTING OF ?
```

Response: MYDISK:=TEXT

Typing this response causes the Filer to list

```
MOLD.TEXT        4  29-JAN-57
NEVERMORE.TEXT  12   5-APR-74
```

USING THE FILER

Type F at the Command level to enter the Filer and the following prompt is displayed:

```
Filer: G(ET, S(AVE, W(HAT, N(EW, L(DIR, R(EM, C(HNG, T(RANS, D(ATE, Q(UIT
or
FILER: G, S, N, L, R, C, T, D, Q
```

Typing ? in response to this prompt displays more Filer commands:

```
FILER: B(AD-BLKS, E(XT-DIR, K(RNCH, M(AKE, P(REFIX, V(OLS, X(AMINE, Z(ERO
or
FILER: W, B, E, K, M, P, V, X, Z
```

The letters and numbers enclosed in brackets which frequently appear at the end of prompt lines indicate the version number of the portion of the program with which you are working.

An individual Filer command is invoked by typing the first letter of the command option as it appears in the prompt line. For example, typing S would invoke the Save command.

In the Filer, answering a Yes/No question with any character other than Y constitutes a "No" answer. With most questions, pressing the RETURN key as your only response will terminate that command and return you to the outer level of the Filer.

For each command requiring a file specification, refer to the file specification diagram earlier in this chapter. In many cases, the entire file specification is not necessary, and in some cases, certain parts of the file specification are not valid. Terminate the specification by pressing the RETURN key. See the required command in the following section for more details.

Whenever a Filer command requests a file specification, you may specify as many files as desired, by separating the file specifications with commas, and terminating this "file list" by pressing the RETURN key. Commands operating on single filenames will keep reading filenames from the file list and operating on them until there are none left. Commands operating on two filenames (such as L(ist-directory, C(hange, and T(ransfer) will take file specifications in pairs and operate on each pair until only one or none remains. If one filename remains, the Filer will prompt for the second member of the pair. If an error is detected at any point in the list, the remainder of the list will be discarded.

THE FILER COMMANDS

34 GENERAL FILE-MOVING COMMAND
34 T(ransfer
42 copying a diskette

44 GENERAL DISKFILE COMMANDS
44 M(ake
45 C(hange
48 R(emove
49 K(runch
50 Z(ero

51 WORKFILE COMMANDS
51 G(et
53 S(ave
55 N(ew
55 W(hat

55 INFORMATION COMMANDS
55 V(olumes
56 L(ist directory
59 E(xtended directory list

61 DISK UPKEEP COMMANDS
61 B(ad blocks
62 X(amine

65 MISCELLANEOUS COMMANDS
65 P(refix
65 D(ate
66 Q(uit

GENERAL FILE-MOVING COMMAND

T(ransfer

Copies the specified file to the given destination.

This command requires that you type two file specifications, one for the source file and one for the destination file, separated with either a comma or a RETURN . Wildcards are permitted, and size specification information is recognized for the destination file.

The source device or diskette must be available (the source diskette must be in one of the disk drives) when the Filer reads your source-file specification, but a destination diskette may be inserted later, in response to a prompting message.

Note: Do not attempt to transfer information between different diskettes having the same volume name. Instead, C(hange the name of one of the diskettes, at least for the duration of the transfer.

EXAMPLE:

Suppose you wish to transfer the file FARKLE.TEXT from the diskette named MYDISK: to the diskette named BACKUP: .

Prompt: TRANSFER WHAT FILE ?   or   TRANSFER ?

Response: MYDISK:FARKLE.TEXT

When you press the RETURN key, the system checks to be sure that the specified source diskette is in one of the disk drives. If MYDISK: is not in any drive, you will see the message

```
MYDISK:FARKLE.TEXT
NO SUCH VOL ON-LINE <SOURCE>
```

If the source diskette is found in a drive, the system then checks to be sure the specified file is on that diskette. If the diskette MYDISK: is in a drive, but it has no file named FARKLE.TEXT , you will see the message

```
MYDISK:FARKLE.TEXT
FILE NOT FOUND <SOURCE>
```

In either case, you will be returned to the outer Filer level. Just insert the correct source diskette in any drive and type T again.

However, let's assume the system succeeds in finding the source diskette and file. The dialogue continues, asking you to specify the destination for the transfer:

Prompt: TO WHERE ?

Response: BACKUP:NAME.TEXT

(Note: The file's source and destination specifications could also have been given both in the first response, separated by a comma.)

When you press the RETURN key, the system checks to be sure the destination diskette is in a disk drive. If it is, the transfer begins. If it is not, there is a pause, and then you will be prompted

```
PUT IN BACKUP:
TYPE <SPACE> TO CONTINUE
```

Put the correct destination diskette in any available drive and press the spacebar. On two-drive systems, you will normally put the source diskette in one drive and the destination diskette in the other drive. On one-drive systems, you will have to remove the source diskette from the disk drive, and then put the destination diskette in the drive.

IMPORTANT: On a single-drive system, DO NOT remove your source diskette until you are prompted to insert the destination diskette. You will be given the following message when it is time to exchange diskettes in the drive:

Prompt:

```
PUT IN BACKUP:
TYPE <SPACE> TO CONTINUE
```

When you see that message, you should remove the source diskette from the drive, insert the correct destination diskette, and press the spacebar. If the specified file is large, you may have to switch the source and destination diskettes several times, until the transfer is completed. Just follow the prompting messages.

Switching the diskettes is not usually necessary on systems with two or more disk drives, as the source and destination diskettes can be in different drives at the same time. However, the one-drive procedure will also work on multiple-drive systems.

When the transfer is complete, the Filer will give you the message

```
MYDISK:FARKLE.TEXT
 --> BACKUP:NAME.TEXT
```

The Filer has made a copy of FARKLE.TEXT as found on the diskette named MYDISK: , and has stored that copy on the diskette BACKUP: under the filename NAME.TEXT .

Note: Once the Filer has been summoned, it resides entirely in the computer's memory. On a one-drive system, you can summon the Filer and then remove the system diskette from the drive in order to insert the source diskette for the Transfer. On a two-drive system, you can summon the Filer, and then remove all system diskettes from the drives in order to use both drives for source and destination diskettes during a transfer. This will save you much unnecessary switching of diskettes when copying a large file. Just remember to replace the system diskette in the boot drive (volume #4:) before Quitting the Filer.

It is often convenient to transfer a file to another diskette without changing its filename. To make this easier, the Filer lets you use the dollar-sign character ($) as a shorthand for "same name", to replace the filename in the destination file specification. In the above example, if you had wished to save the file FARKLE.TEXT on diskette BACKUP: under the same filename FARKLE.TEXT, you could have typed:

```
MYDISK:FARKLE.TEXT,BACKUP:$
```

WARNING: Avoid typing the second file specification with the filename completely omitted! For example, a response to the Transfer prompt of the form:

```
MYDISK:FARKLE.TEXT,BACKUP:
```

generates the message:

```
DESTROY BACKUP: ?
```

Typing the response Y causes the directory of BACKUP to be wiped out!

A file can be transferred from a diskette to a different place on the same diskette by giving the same volume name for both source and destination file specifications. This is frequently useful when the you wish to relocate a file on the diskette. Specifying the number of blocks desired will cause the Filer to copy the file in the first (lowest block numbers) unused diskette area of at least that size. If no size specification is given, the file is always written in the largest unused area.

If you specify the same filename for both source and destination on a same-diskette transfer, then the Filer rewrites the file to the size-specified area (the largest unused area, if not specified), and removes the original file.

EXAMPLE:

Prompt: TRANSFER WHAT FILE ? or TRANSFER ?

Response: MYDISK:QUIZZES.TEXT,MYDISK:$[20]

Typing this response would cause the Filer to rewrite QUIZZES.TEXT on MYDISK: in the first area of at least 20 blocks (looking from block 0) and then to remove the previous version of QUIZZES.TEXT .

Note: do not use this feature for re-naming a file. The C(hange command is designed for that purpose, and is less risky.

If you give the same volume NUMBER for both source and destination file specification, the system assumes you are going to change diskettes in that drive. You will see the message

```
INSERT DESTINATION DISK
TYPE <SPACE> TO CONTINUE
```

The same assumption is made any time you give a volume number, in the destination file specification, that specifies the same drive occupied by the source diskette.

Files may be transferred to such volumes as CONSOLE: (for a quick screen listing of a file) and PRINTER: (to print a file), as well as to a disk, by specifying the appropriate volume name or number (see VOLUMES, earlier in this chapter) in the destination file specification. A filename on a device other than a disk is ignored.

It is generally a good idea to make certain that a non-disk device is on-line (actively connected to the system and turned on) when you attempt a Transfer to that device. If it isn't, the system may "hang", and you will have to press the RESET key to recover.

EXAMPLE:

Prompt: TRANSFER WHAT FILE ? or TRANSFER ?

Response: FARKLE.TEXT

Prompt: TO WHERE ?

Response: PRINTER:

Typing the above responses will cause the file FARKLE.TEXT , as found on the Prefix diskette volume, to be printed (assuming a printer is properly connected to your system).

You may also transfer from input devices other than disks, such as the keyboard. Filenames accompanying a non-disk volume name or number are ignored.

EXAMPLE:

Prompt: TRANSFER WHAT FILE ? or TRANSFER ?

Response: CONSOLE:

Prompt: TO WHERE ?

Response: PRINTER:

After these responses, you can use your keyboard as a typewriter. Nothing will appear on the printer until you type the "End-Of-File" character, CTRL-C (Note some printers require you to press the RETURN key before pressing CTRL-C). Then all your typing will be sent to the printer.

The wildcard capability is allowed in the T(ransfer command. When using wildcards, the subset-specifying strings in the source filenames will be replaced by the analogous strings (called "replacement strings") in the destination filenames. Any of the subset-specifying or replacement strings may be empty. The portion of each source filename accounted for by the = or ? wildcard character is reproduced unchanged in the corresponding destination filename. Remember that the Filer considers the one-character, wild-card-alone file specification (= or ?) to specify every file on the volume.

EXAMPLE:

Suppose the Prefix diskette volume MYDISK: contains these files:

```
PAUCITY
PARITY
PENALTY
```

Further, suppose the destination diskette is named ODDNAMZ:

Prompt: `TRANSFER WHAT FILE ?` or `TRANSFER ?`

Response: `P=TY,ODDNAMZ:V=S`

Typing this response would cause the Filer to reply:

```
MYDISK:PAUCITY
 --> ODDNAMZ:VAUCIS
MYDISK:PARITY
 --> ODDNAMZ:VARIS
MYDISK:PENALTY
 --> ODDNAMZ:VENALS
```

EXAMPLE:

Suppose the Prefix diskette volume MYDISK: contains these files:

```
CHAP1.TEXT
CHAP2.TEXT
CHAPTER-3.TEXT
CHAP4.TEXT
```

Further, suppose the destination diskette is named BACKUP:

Prompt: `TRANSFER WHAT FILE ?` or `TRANSFER ?`

Response: `C=XT`

Prompt: `TO WHERE ?`

Response: `BACKUP:OLDC=XT`

Typing these responses would cause the Filer to reply:

```
MYDISK:CHAP1.TEXT
 --> BACKUP:OLDCHAP1.TEXT
MYDISK:CHAP2.TEXT
 --> BACKUP:OLDCHAP2.TEXT
MYDISK:CHAPTER-3.TEXT
 --> NOT PROCESSED
MYDISK:CHAP4.TEXT
 --> BACKUP:OLDCHAP4.TEXT
```

On the third attempted transfer, the destination filename would have been OLDCHAPTER-3.TEXT, which exceeds the 15-character limit for filenames. Therefore, that file was "NOT PROCESSED" . If all of the destination filenames exceed 15 characters, each source file is marked "NOT PROCESSED" and this additional message is given:

```
BAD DEST FOR FILES FOUND
```

Using the single character = as the source filename specification will cause the Filer to attempt to transfer every file on the source diskette, adding these files to the information that was already stored on the destination diskette. You can use this feature to transfer all the information on one diskette to another diskette, withOUT destroying any information already on the destination diskette. (If you wish to make an exact copy of the source diskette, completely erasing any information that was formerly stored on the destination diskette, please refer to the material later in this section on volume-to-volume transfers: COPYING A DISKETTE .)

Using the single character = as the destination filename specification will have the effect of replacing any subset-specifying strings in the source specification with nothing.

A brief reminder: in any wildcard specification, the single character ? may be used in place of = . The only difference is that a ? in either specification (or both) causes the Filer to ask you for verification before each file is transferred. This takes somewhat longer, but you are more certain of transferring only the files you intended to transfer.

A source or a destination file specification must contain only one wildcard character. A specification such as

```
MYDISK:?UGH?
```

is NOT a legal specification. An attempt to use such a specification as either the source or the destination of a transfer will cause the jargon message

```
SCAN STRING - ILLEGAL FORMAT
```

If the source file specification contains a wildcard character, and the destination device is a disk, then the destination file specification must also contain a wildcard character. If you fail to have a wildcard character in both source and destination specification (it need not be the same wildcard), you are given the message

```
BAD FORM (WILD <TO> NON-WILD) CARD
```

and your transfer is terminated. The only exception to this occurs when you use the dollar sign ($) shorthand for the destination file specification.

EXAMPLE:

Suppose the diskette MYDISK: contains the following files:

```
CHAPTER1.TEXT
CHAPTER14B.TEXT
INTRO.TEXT
```

Further, suppose you wish to transfer the files CHAPTER1.TEXT and INTRO.TEXT to the diskette BACKUP: , retaining the same file names on the backup diskette.

Prompt: `TRANSFER ?`

Response: `MYDISK:?.TEXT,BACKUP:$`

Typing this response would cause the screen to clear, and then the following message would appear:

```
TRANSFER CHAPTER1.TEXT ?
```

Since you wish to transfer CHAPTER1.TEXT , type a Y for "Yes". A copy of the file CHAPTER1.TEXT would then be transferred from MYDISK: to BACKUP: , and the Filer would proceed to ask if you wish to transfer the next file whose name ends in .TEXT . The complete dialogue might appear as follows:

```
TRANSFER CHAPTER1.TEXT ? Y
MYDISK:CHAPTER1.TEXT
 --> BACKUP:CHAPTER1.TEXT
TRANSFER CHAPTER14B.TEXT ? N
TRANSFER INTRO.TEXT ? Y
MYDISK:INTRO.TEXT
 --> BACKUP:INTRO.TEXT
```

Instead of a "Y" or "N" response, you may press the ESC key. This will return you to the outer level of the Filer.

Copying a Diskette

You can copy an entire diskette. The file specifications for the source and for the destination should each consist of a disk volume name or number only. This method of transferring the contents of a source diskette volume onto a destination diskette volume erases any previous contents that were on the destination diskette so that it becomes an exact, literal copy of the source diskette. After copying, the destination diskette has the same volume name as the source diskette.

EXAMPLE:

Suppose you desire an extra copy of the diskette MYDISK: and you are willing to sacrifice diskette EXTRA:

Prompt: TRANSFER WHAT FILE ? or TRANSFER ?

Response: MYDISK:,EXTRA:

Prompt: TRANSFER 280 BLOCKS ? (Y/N)

Response: Y

Note: To copy an entire diskette (each diskette used by the Apple Pascal system contains 280 blocks), you will always type the response Y . Each diskette's directory tells your computer how many blocks are on that diskette. If your system ever gives a message such as

```
TRANSFER 1100 BLOCKS? (Y/N)
```

(or any number other than 280 blocks), the diskette's directory is probably damaged.

Prompt: DESTROY EXTRA: ?

WARNING: If you type Y , the directory (and therefore, your access to the contents) of EXTRA: will be destroyed! The diskette named EXTRA: will then become an exact copy of MYDISK: , even having the same volume name. Often this is desirable for backup purposes, since it is relatively easy to copy a diskette this way, and the volume name can be changed (see the C(hange command) if desired. An N response will return you to the outer level of the Filer.

Although it is certainly possible to transfer one diskette volume to another using a single-disk-drive system, it is a fairly tedious process, since a great deal of diskette exchanging is necessary for the complete transfer to take place.

IMPORTANT: On a single-drive system, DO NOT remove your source diskette until you are prompted to insert the destination diskette. You will be given the following message when it is time to exchange diskettes in the drive:

```
Prompt:    PUT IN EXTRA:
           TYPE <SPACE> TO CONTINUE
```

You should now remove the source diskette from the drive, insert the correct destination diskette, and press the spacebar. The Filer will soon tell you

```
Prompt:    PUT IN MYDISK:
           TYPE <SPACE> TO CONTINUE
```

and so on, back and forth, until you have exchanged the source and destination diskettes in the drive about 20 times. Finally, the Filer will give you the welcome message:

```
           MYDISK:          --> EXTRA:
```

to tell you the transfer is completed.

Note: Once the Filer has been summoned, it resides entirely in the computer's memory.

One-drive note: On a one-drive system, you can summon the Filer and then remove the system diskette from the drive in order to insert the source diskette for the transfer. Just remember to replace the system diskette in the boot drive (volume #4:) before Quitting the Filer.

Two-drive note: On a two-drive system, you can summon the Filer, and then remove all system diskettes from the drives in order to use both drives for source and destination diskettes during a transfer. This will save you much unnecessary switching of diskettes when copying an entire diskette. Again, remember to replace the system diskette in the boot drive (volume #4:) before Quitting the Filer.

Another one-drive note: One-drive users cannot make a volume-to-volume copy onto a destination diskette that has the same volume name as the source diskette. Instead, before trying to make the copy, use the Filer's C(hange command to change the volume name of either diskette, or use the Z(ero command to rename the destination diskette while erasing its directory.

Another multi-drive note: In multiple-drive systems, the source and destination diskettes are usually placed in different drives. If the two diskettes have the same volume name, you can refer to each diskette by its drive volume number, instead of by name. By their different drive numbers shall the Filer know them. Note that this full-disk copy is an exception to the rule which forbids two diskettes with the same name in the system.

GENERAL DISKFILE COMMANDS

M(ake

Creates a diskette directory entry with the specified filename.

This command requires you to type a file specification. Wildcard characters are not allowed. The file size specification option is extremely helpful, since, if it is omitted, the Filer creates the specified file by consuming the largest unused area of the disk. The file size is determined by following the filename with the desired number of blocks, enclosed in square brackets [and] . Some special cases are:

[0] - Equivalent to omitting the size specification. The file is created using all of the largest unused area.

[*] - The file is created using all of the second largest area, or half of the largest area, whichever is larger.

Files with filenames ending in .TEXT must occupy at least four blocks, and must occupy an even number of blocks (see this manual's appendix FILE FORMATS for details). An attempt to M(ake a .TEXT file with fewer than four blocks results in the message NO ROOM ON VOL. If you M(ake a .TEXT file specifying an odd number of blocks, the file will actually be made with one fewer block.

EXAMPLE:

```
Prompt:   MAKE WHAT FILE?

Response: MYDISK:FARKLE.TEXT[28]
```

This response creates the dummy file FARKLE.TEXT on the volume MYDISK: in the first unused 28-block area encountered.

The M(ake command is commonly used to reserve an area on the disk for some future use. The created file's name may serve as a reminder that you need to write a section by that name, and save you some space on the diskette to do so. It also prevents use of that space by other files.

When you make a file, you simply create a diskette directory entry, without in any way changing the actual information stored on the portion of the diskette to which that directory entry refers. If you forget that the file is a "dummy" file, you can G(et the file (if it ends in .TEXT) and attempt to read into the Editor whatever information may have been stored on the diskette in that location. Usually, this will just be nonsense, or part of some file you never wanted to see again, but occasionally it can be useful.

Suppose you have just R(emoved a 19 block file, which started at block 134. An E(xtended directory list of the diskette may show the "hole" where that file used to be, as a 19 block <unused> area starting at block 134. If you can now M(ake a file (of any name) that exactly occupies the blocks the R(emoved file occupied, the new file will contain exactly the same information the Removed file contained. Thus, if you know enough information about the location of a file before it was Removed, and if nothing has been written over that area of the diskette since the removal, you can sometimes use the M(ake command to recover a Removed file.

C(hange

Changes a diskette file′s filename, or changes a diskette′s volume name.

This command requires two file specifications. The first of these specifies the file or the volume whose name is to be changed; the second specification shows the new filename or the new volume name. The first specification is separated from the second specification either by a comma (,) or by pressing the RETURN key. If the first file specification contains a filename, any volume name or number in the second file specification is ignored, since obviously the "old file" and the "new file" are on the same volume! Size specification information is ignored.

If you change the name of the Prefix diskette, the volume name that the system supplies as Prefix is also changed. Similarly, if you change the name of the "system" or "root" diskette, the name that the system supplies for the asterisk (*) volume-specifier is also changed.
EXAMPLE:

The file F5.TEXT is on the diskette in disk drive volume #5:

Prompt: CHANGE WHAT FILE ? or CHANGE ?

Response: #5:F5.TEXT

When you press the RETURN key, the dialogue continues:

Prompt: CHANGE TO WHAT ?

Response: HOOHAH

Typing the above response changes the name in the directory from F5.TEXT to HOOHAH . Filetypes (such as TEXTFILE or CODEFILE) are originally determined by the filename′s suffix (such as .TEXT or .CODE). The C(hange command does not affect the filetype, but it also does not automatically place the correct standard filetype suffix after the new filename. In the above case, HOOHAH would still be listed as type TEXTFILE by an E(xtended directory list. However, since the G(et command searches for the suffix .TEXT in order to identify a textfile as the workfile, you would have to C(hange the filename HOOHAH to HOOHAH.TEXT before that file could be used as the workfile.

Wildcard specifications are legal in the C(hange command. If a wildcard character is used in the first file specification, then a wildcard must be used in the second file specification. The subset-specifying strings in the first file specification are replaced by the analogous strings (henceforward called replacement strings) given in the second file specification. The Filer will not change the filename if the change would have the effect of making the filename too long (more than 15 characters).

EXAMPLE:

The diskette named MYDISK: contains these files:

```
CHAP1.TEXT
CHAP2.TEXT
CHAPTER-3.TEXT
CHAP4.TEXT
```

Prompt: CHANGE WHAT FILE ?   or   CHANGE ?

Response: MYDISK:C=XT,OLDC=XT

After you typed the above response (the two parts of the response were separated by a comma, this time, but you could also press the RETURN key to separate the responses), the Filer would then indicate the following name changes. Only the files' filenames are changed; the contents of the files themselves are left unmodified.

```
MYDISK:CHAP1.TEXT
  --> OLDCHAP1.TEXT
MYDISK:CHAP2.TEXT
  --> OLDCHAP2.TEXT
MYDISK:CHAPTER-3.TEXT
  --> NOT PROCESSED
MYDISK:CHAP4.TEXT
  --> OLDCHAP4.TEXT
```

In the third attempted name change, the "destination" filename would have been OLDCHAPTER-3.TEXT, which exceeds the 15-character limit for filenames. Therefore, that file was "NOT PROCESSED" . If all of the "destination" filenames exceed 15 characters, this additional message is given:

```
BAD DEST FOR FILES FOUND
```

The subset-specifying strings may be empty, as may the replacement strings. The Filer considers the one-character file specification = (where both subset-specifying strings are empty) to specify every file on the diskette.

EXAMPLES:

Prompt: `CHANGE WHAT FILE ?` or `CHANGE ?`

Response#1: `=,Z=Z`

Typing this response would cause every filename on the Prefix diskette to have a Z added before the first character and after the last character.

Response#2: `Z=Z,=`

Typing this response would (again, on the Prefix diskette) erase the terminal and initial Z from each filename that possessed both.

Suppose the Prefix diskette contained these filenames:

```
THIS.TEXT
THAT.TEXT
```

Response#3: `T=T,=`

Typing response #3 would result in changing THIS.TEXT to HIS.TEX , and THAT.TEXT to HAT.TEX .

Response#4: `=.TEXT,OLD.=`

Typing this response would add the prefix OLD. to every filename on the Prefix diskette, and remove the suffix .TEXT from every filename. CHAP1.TEXT would thus become OLD.CHAP1 , and CHAP2.TEXT would become OLD.CHAP2 . This would quickly mark all your old versions of a file and simultaneously make those versions safe from accidental access by the G(et command.

The diskette's volume name may also be changed, by specifying the current diskette volume name or number and (after a comma or RETURN) a new volume name for the diskette.

EXAMPLE:

Prompt: `CHANGE WHAT FILE ?` or `CHANGE ?`

Response: `NOTSANE:,WRKDISK:`

Typing this response would cause the system to give this message:

```
NOTSANE:                        --> WRKDISK:
```

showing that the diskette named NOTSANE: has been renamed WRKDISK:.

R(emove

Removes file entries from the directory, which makes those diskette files inaccessible. While a removed file's contents are still stored on the diskette, and it may sometimes be possible to recover them in an emergency (see M(ake), the system acts as if a removed file had been erased from the diskette. That area of the diskette is then considered free for overwriting with other files.

This command requires one file specification for each diskette file that you wish to remove. Wildcards are legal. Size specification information is ignored.

EXAMPLE:

Suppose the Prefix diskette contains these files:

```
AARDVARK.TEXT
ANDROID.CODE
QUINT.TEXT
AMAZING.CODE
```

Prompt: REMOVE WHAT FILE ?   or   REMOVE ?

Response: AMAZING.CODE

Typing this response tells the system to remove the file AMAZING.CODE from the Prefix diskette's directory. The system then considers that file erased from the diskette, although only the directory has been changed.

To remove SYSTEM.WRK.TEXT and/or SYSTEM.WRK.CODE , the Filer's N(ew command should be used, or the system may get confused. IT IS VERY IMPORTANT TO REMEMBER THIS LITTLE QUIRK, BECAUSE THE SYSTEM WON'T WARN YOU.

As noted before, wildcard removes are legal. Fortunately, before finalizing any wildcard removes, the Filer asks if you wish to

```
UPDATE DIRECTORY ?
```

Typing Y in response to this prompt causes all the specified files to be removed. Typing N returns you to the outer level of the Filer without any removes having occurred.

EXAMPLE:

Prompt: REMOVE WHAT FILE ?   or   REMOVE ?

Response: A=CODE

Typing this response causes the Filer to remove AMAZING.CODE and ANDROID.CODE from the Prefix diskette directory.

WARNING: Remember that the Filer considers the one-character file specification = (where both subset-specifying strings are empty) to specify every file on the volume. Typing an = alone will cause the Filer to remove every file on your directory!

K(runch

Moves the files on the specified diskette volume so that unused blocks are combined. You use this when you run out of space or seem about to, because the unused space on the diskette is fragmented. Using the E(xtended directory list command to list the directory will show you how the unused space is distributed on the diskette. After typing K, all the unused space will be together at the end of the diskette (or at some other place on the disk, specified by you).

This command requires that you type a diskette volume name or number. The specified diskette volume must be on-line (currently available to the system). It is a good idea to perform a bad block scan of the volume before K(runching, to avoid writing files over bad areas of the diskette. If bad blocks are encountered, they must be either fixed or marked before the K(runch (see X(amine).

As each file is moved, its name is displayed on the screen. If SYSTEM.PASCAL is moved, the system must be reinitialized by booting.

WARNING! Do not touch the disk, the RESET key, the power switch or the disk-drive door until K(runch tells you it has completed its task. To do otherwise may make the information on your diskette unreadable.

EXAMPLE:

Suppose you wish to K(runch the boot diskette:

Prompt: CRUNCH WHAT VOL ? or CRUNCH ?

Response: *

You could also have responded with the volume number (#4:) or the volume name of the boot diskette, of course.

Prompt: FROM END OF DISK, BLOCK 280 ? (Y/N)

Typing the response Y initiates the normal K(runch. Typing an N will cause the prompt:

Prompt: STARTING AT BLOCK # ?

If you type a block number in response to this prompt, the Filer will attempt to make room for new files in the area surrounding the block number that you specified. It does this by moving files forward (toward lower block numbers) which are below the specified block, and moving files backward (toward higher block numbers) which are above the specified block. This feature allows you to re-arrange files, by placing them at diskette locations other than the end of the diskette.

Note: If you specify a Krunch starting block that is clearly within an existing file, but the Filer tells you the diskette is already Krunched, just try again with a starting block in the next higher-block-numbered file.

Z(ero

"Erases" the directory of the specified volume (by writing zeros into it). The previous directory is rendered irretrievable. This is used to "recycle" a used diskette; the system forgets anything previously stored on the diskette and the diskette is ready to be used again. This command does NOT format the diskette: the diskette must already have been formatted by eXecuting the FORMATTER utility program (see this manual′s chapter UTILITY PROGRAMS).

EXAMPLE:

Suppose you wish to forget all information stored on a diskette named OLDDISK, in disk drive volume #5: , in order to re-use it as a clean, blank diskette.

Prompt: `ZERO DIR OF WHAT VOL ?` or `ZERO DIR OF ?`

Response: `#5:`

Prompt: `DESTROY OLDDISK: ?`

Response: `Y`

Prompt: `DUPLICATE DIR ?`

Always respond to this prompt by typing an N response, which will cause the usual single directory to be maintained. The Apple Pascal system does not support duplicate directories. Next you will see:

Prompt: `ARE THERE 280 BLKS ON THE DISK ? (Y/N)`

Response: `Y`

The Apple Pascal system uses only 280-block diskettes, so your answer to this prompt should always be Y for "Yes". Each diskette's directory contains the number of the blocks on the diskette. If your system ever asks

```
ARE THERE 1100 BLKS ON THE DISK ? (Y/N)
```

(or any number other than 280), the diskette's directory is probably damaged.

Now you will be asked to name the newly-zeroed diskette:

Prompt: `NEW VOL NAME ?`

Response: `NEWDISK:`

(or you can type any other valid volume name); and then you will be asked to verify the new name.

Prompt: `NEWDISK: CORRECT ?`

Response: `Y`

Typing a Y response to this prompt causes the Filer to respond with the message:

```
NEWDISK: ZEROED
```

WORKFILE COMMANDS

G(et

Identifies the designated diskette file for later use as the workfile. The next time you attempt to Edit, Compile, or Run, the designated file will be used. At that time, if the designated file is no longer available to the system this message is given:

```
ERROR: WORKFILE LOST.
```

Note: Although you are told that the specified file has been "loaded", this command does NOT actually transfer the specified file to the boot diskette file named SYSTEM.WRK (or to any other file). The file SYSTEM.WRK is usually created by Updating the workfile from the Editor (see this manual's chapter THE COMMAND LEVEL for more information about the workfile, and the chapter THE EDITOR for more information about the Editor).

One-drive note: In one-drive systems, since your boot diskette (usually APPLE0:) must be in the drive to Edit, Compile, and Run the designated workfile, you can only effectively G(et files that you have previously T(ransferred to your boot diskette.

If there is already a workfile SYSTEM.WRK present on the boot diskette when you issue the G(et command, you are prompted:

```
            THROW AWAY CURRENT WORKFILE ?
```

Response: Y will clear the workfile, removing all files SYSTEM.WRK from the boot diskette, while N returns you to the outer level of the Filer.

Typing the filename's suffix in the file specification is not necessary. If the volume name of the diskette is not given, the Prefix diskette is assumed. Wildcards are not allowed, and the size specification option is ignored.

EXAMPLE:

Suppose the Prefix diskette contains the following files:

```
            FILERDOC2.TEXT
            ABSURD.CODE
            HYTYPER.CODE
            STASIS.TEXT
            LETTER1.TEXT
            FILER.DOC.TEXT
            STASIS.CODE
```

Prompt: GET WHAT FILE ? or GET ?

Response: STASIS

The Filer responds with the message

```
            TEXT & CODE FILE LOADED
```

since both text and code file exist. Had you typed STASIS.TEXT or STASIS.CODE , the result would have been the same: both text and code versions would have been identified for later use as the workfile. If only one of the versions exists, as in the case of ABSURD.CODE, then that version is identified for later workfile use, regardless of whether text or code was requested. Typing ABSURD.TEXT in response to the prompt would generate the message: CODE FILE LOADED . Working with the workfile may create a number of files whose names begin SYSTEM.WRK. , as parts of the workfile. These files will disappear when the S(ave command is used to save the contents of the workfile under their original filename or under a new filename. If the system is rebooted before the S(ave command is used, the original name of the workfile's contents (as specified by the G(et command) will be forgotten.

S(ave

Saves all versions of the boot diskette's workfile SYSTEM.WRK under the filename originally specified with G(et or under a different filename which you specify.

If a file already exists with the specified filename and if you are S(aving your file onto a diskette other than the boot diskette, you are asked for verification before the old file is removed. In that case, the workfile is saved under the specified name only after the old file has been removed.

If you are S(aving the workfile as another filename on the boot diskette, the workfile (which is already on that diskette) is simply renamed. When you S(ave the workfile on a diskette other than the boot diskette, the system is actually performing a T(ransfer of the workfile. Thus the workfile is unchanged after the S(ave is completed.

The entire file specification is not necessary. In particular, DO NOT specify a suffix. The correct suffix for each version of the workfile (.TEXT, .CODE, etc.) is supplied automatically, in addition to any suffix that you might type. Unlike many parts of the system, ending the specified filename with a period does NOT suppress the addition of a suffix. If the diskette volume name or number is not given, the Prefix diskette is assumed. Wildcards are not allowed, and the size specification option is ignored.

One-drive note: Even on one-drive systems, S(ave works just fine if there is only one version of the workfile. If you S(ave the workfile onto a diskette other than the boot diskette, the Filer will prompt you to put the destination diskette into the drive at the correct time. However, only the FIRST version of the workfile (usually .TEXT) is S(aved onto the destination diskette. If the Filer returns to the boot diskette to get another version of the workfile (.CODE, say), the boot diskette is not in the drive and the command is terminated. If you want to S(ave MORE THAN ONE version of the workfile (.TEXT and .CODE, say), first S(ave the workfile onto the boot diskette. This renames the workfile versions and tells the system that your workfile is gone. THEN T(ransfer the S(aved versions onto your destination diskette, one version at a time.

EXAMPLE:

Prompt: SAVE AS MYDISK:OLDFILE ?

Response: N

Prompt: SAVE AS WHAT FILE ? or SAVE AS ?

Response: Type a filename of 10 or fewer characters, observing the filename conventions discussed under FILES (earlier in this chapter). This causes the Filer to remove (after asking you for verification) any old file having the specified filename, and then to save the workfile under that name. For example, typing X in response to the prompt causes the workfile to be saved on the Prefix disk as X.TEXT . If a codefile has been compiled or assembled since the last update of the workfile, that codefile will also be saved, as X.CODE .

The Filer automatically appends the suffixes .TEXT and .CODE to files of the appropriate type. Explicitly typing AFILE.TEXT in response to the prompt will cause the Filer to save this file as AFILE.TEXT.TEXT .

EXAMPLE:

Prompt: SAVE AS WHAT FILE ?   or   SAVE AS ?

Response: RED:EYE

If one of your disk drives contains a diskette named RED: , you will soon see the message

```
APPLE1:SYSTEM.WRK.TEXT
 --> RED:EYE.TEXT
```

This message tells you that the workfile named SYSTEM.WRK.TEXT , on the boot diskette named APPLE1: , has been successfully transferred to the file named EYE.TEXT , on the diskette named RED: . If there is no diskette named RED in any disk drive, you will see the message

```
PUT IN RED:
TYPE <SPACE> TO CONTINUE
```

This gives you the chance to insert a diskette named RED , if you have one, into a disk drive. RED:EYE constitutes a file specification, and this response will tell the Filer to attempt to transfer the workfile to the specified volume and file (see the T(ransfer command). If you specified diskette RED: by accident, press the spacebar anyway. The system will not find diskette RED: , and the command will be terminated.

EXAMPLE:

Suppose you earlier used the G(et command to designate the file MYDISK:LETTER as the next workfile. You then Q(uit the Filer and entered the E(ditor, causing MYDISK:LETTER.TEXT to be read into the computer. Finally, you added some new material to the file, and then used the Editor's Q(uit and U(pdate commands to store the new version of the file on the boot diskette as SYSTEM.WRK.TEXT .

Now, back in the Filer again, you type S for S(ave and receive this prompt:

```
SAVE AS MYDISK:LETTER ?
```

If you type a Y , the Filer first asks

```
REMOVE OLD MYDISK:LETTER.TEXT ?
```

Typing another Y causes your previous version of LETTER.TEXT to be removed from MYDISK: , and then causes the new version (stored as APPLE1:SYSTEM.WRK.TEXT) to be saved on MYDISK:

```
          APPLE1:SYSTEM.WRK.TEXT
           --> MYDISK:LETTER.TEXT
```

N(ew

Clears the workfile, so that there is no default file to be used automatically by E(dit, C(ompile, A(ssemble, and R(un . The last file specified as the workfile by the Filer's G(et command is no longer so designated. All versions of the workfile SYSTEM.WRK saved on the boot diskette are removed from the directory (SYSTEM.LST.TEXT is also removed). There will be no workfile on the boot diskette until a workfile is saved (usually using the Editor's U(pdate command).

If there is already a workfile SYSTEM.WRK present on the boot diskette when you issue the N(ew command, you are prompted:

```
          THROW AWAY CURRENT WORKFILE ?
```

Response: Y will clear the workfile, removing all files SYSTEM.WRK from the boot diskette, while N returns you to the outer level of the Filer.

Use the N(ew command to clear away the automatically-loaded workfile before you try to create a new file in the Editor or Compile any file other than the workfile.

W(hat

Identifies the name and state (saved or not) of the workfile.

If the workfile has been S(aved onto any diskette other than the boot diskette, the W(hat command continues to report the workfile as (NOT SAVED). This is because the workfile still exists on the boot diskette.

INFORMATION COMMANDS

V(olumes

Lists the input and output volumes (devices or diskettes) currently "on-line" (actively connected into the system), by volume name and by volume number.

A typical display for a single-drive system, with few peripherals, might be:

```
VOLS ON-LINE:
  1   CONSOLE:
  2   SYSTERM:
  4 # APPLE0:
ROOT VOL IS - APPLE0:
PREFIX IS   - APPLE0:
```

The volumes CONSOLE: and SYSTERM: are always available. They are just two different ways to refer to the screen and the keyboard.

A four-drive system, with a printer and with a modem for communicating over telephone lines, might give a display like this:

```
VOLS ON-LINE:
  1   CONSOLE:
  2   SYSTERM:
  4 # APPLE1:
  5 # APPLE2:
  6   PRINTER:
  7   REMIN:
  8   REMOUT:
 11 # APPLE3:
 12 # JEF:
ROOT VOL IS - APPLE1:
PREFIX IS   - JEF:
```

The boot diskette, also called the system volume, is indicated here as the ROOT VOL -- in this case the diskette named APPLE1: , in disk drive volume #4: . The default volume is indicated here as the PREFIX volume -- in this case it has been changed by the P(refix command to JEF: , the name of the diskette in disk drive volume #12: . In general, the Prefix volume will be the same as the boot volume unless the Prefix (see the P(refix command) has been changed. Block-structured devices (disks) are indicated by a "pound" sign (#).

L(ist Directory

Lists a diskette's directory, or part of one, to the volume and file specified (default is CONSOLE:). All files and unused areas are listed along with their block length and last modification date.

You may list any portion of the directory, using the "wildcard" option, and may also write the directory, or any portion of it, to a volume or filename other than CONSOLE: . This is why you must sometimes give both a source file specification and a destination file specification. The destination file specification should NOT include a wildcard.

Source file specification consists of a mandatory disk volume name or number, and optional wildcard and subset-specifying strings, which may be empty. The source file specification must be separated from a destination file specification by a comma (,). Destination file specification consists of a volume name or number and, if the volume is a disk, you MUST include a filename.

A directory listing stops when it has filled the screen. Press the spacebar to continue the listing, or press the ESC key to abandon the listing and return to the Filer prompt line.

EXAMPLE:

The most frequent use of this command is to list an entire diskette directory on the screen. The following display, which represents a complete directory listing for an example diskette APPLE0: , could be generated by typing any valid volume name or number for APPLE0: in response to the prompt,

Prompt: DIR LISTING OF WHAT VOL ? or DIR LISTING OF ?

Response: #4:

```
APPLE0:
SYSTEM.PASCAL       36    4-MAY-79
SYSTEM.MISCINFO      1    4-MAY-79
SYSTEM.COMPILER     71   30-MAY-79
SYSTEM.EDITOR       45   29-JAN-79
SYSTEM.FILER        28   24-MAY-79
SYSTEM.LIBRARY      36   22-JUN-79
SYSTEM.CHARSET       2   14-JUN-79
SYSTEM.SYNTAX       14   18-APR-79
TUNAFISH.TEXT        4    8-JUL-79
SYSTEM.WRK.TEXT      4   17-JUL-79
SYSTEM.WRK.CODE      2   17-JUL-79
11/11 FILES, 31 UNUSED, 23 IN LARGEST
```

The bottom line of the display informs you that 11 files out of a total of 11 files on the diskette have been listed, that there are 31 out of a total of 280 diskette blocks left for you to use, and that there are 23 contiguous blocks in the largest unused area on the diskette. The first ratio shows that you are looking at a complete listing of the diskette's directory, and not a partial listing as discussed below. The last number shows the size of the largest file that you could now store onto this diskette. Even though there are 31 unused blocks available on the diskette, the largest file you could store would be 23 blocks, because a file must be stored in contiguous blocks.

EXAMPLE:

Here is a L(ist-directory transaction involving wildcards.

Prompt: DIR LISTING OF WHAT VOL ? or DIR LISTING OF ?

Response: #4:S=R

Typing the response above might generate the following display:

```
APPLE0:
SYSTEM.COMPILER    71   30-MAY-79
SYSTEM.EDITOR      45   29-JAN-79
SYSTEM.FILER       28   24-MAY-79
3/11 FILES, 93 UNUSED, 93 IN LARGEST
```

A partial listing of a directory assumes that the last file listed is the last file on the diskette, and uses that assumption in calculating the number of unused blocks remaining on the diskette beyond the last listed file. This faulty assumption usually gives an incorrect number of unused blocks, and an incorrect size for the largest unused area. This is only a problem on partial listings; complete listings give the correct numbers.

EXAMPLE:

This L(ist-directory transaction involves writing a subset of the directory to a device other than the default CONSOLE:

Prompt: DIR LISTING OF WHAT VOL ? or DIR LISTING OF ?

Response: `APPLE0:S=R,PRINTER:`

Typing the above response causes this message:

```
APPLE0:
SYSTEM.COMPILER    71    4-MAY-79
SYSTEM.EDITOR      45   29-JAN-79
SYSTEM.FILER       28   24-MAY-79
3/11 FILES, 93 UNUSED, 93 IN LARGEST
```

to appear on the printer (if you have a printer, and if it was turned on!). It's that easy. The number of unused blocks is still wrong, since this is a partial directory listing.

EXAMPLE:

This L(ist-directory transaction involves writing the directory to a disk:

Prompt: DIR LISTING OF WHAT VOL ? or DIR LISTING OF ?

Response: `#4:,#4:DRCTRY.TEXT`

After typing this response, you will see the message

```
WRITING.........
```

as the Filer creates the file DRCTRY.TEXT on the diskette in disk drive #4: . This file would contain the entire directory of the diskette in drive #4: , as it looked BEFORE the file DRCTRY.TEXT was added to it.

E(xtended Directory List

Lists the directory of a diskette, giving more detail than the L(ist-directory command. All files and unused areas are listed along with (in this order) their block length, last modification date, the starting block address, and the filetype.

This command takes a little longer than the L(ist-directory command, but it gives important extra information about the distribution of files on your diskette.

The prompt lines, syntax, and wildcard options are exactly the same for this command as for the L(ist-directory command discussed above. For more details and examples, look at the L(ist-directory discussion.

EXAMPLE:

An example display for a diskette APPLE0: , also shown in the previous L(ist-directory discussion, is shown below.

Prompt: `DIR LISTING OF WHAT VOL ?` or `DIR LISTING OF ?`

Response: `#4:`

```
APPLE0:
SYSTEM.PASCAL      36   4-MAY-79    6  DATA
SYSTEM.MISCINFO     1   4-MAY-79   42  DATA
SYSTEM.COMPILER    71  30-MAY-79   43  CODE
SYSTEM.EDITOR      45  29-JAN-79  114  CODE
SYSTEM.FILER       28  24-MAY-79  159  CODE
SYSTEM.LIBRARY     36  22-JUN-79  187  DATA
SYSTEM.CHARSET      2  14-JUN-79  223  DATA
SYSTEM.SYNTAX      14  18-APR-79  225  TEXT
< UNUSED >          4             239
TUNAFISH.TEXT       4   8-JUL-79  243  TEXT
< UNUSED >          4             247
SYSTEM.WRK.TEXT     4  17-JUL-79  251  TEXT
SYSTEM.WRK.CODE     2  17-JUL-79  255  CODE
DRCTRY.TEXT         4  18-JUL-79  257  TEXT
< UNUSED >         19             261
12/12 FILES, 27 UNUSED, 19 IN LARGEST
```

If you are using the Apple Pascal system with an external terminal whose screen shows an 80-character-wide, upper-and-lower-case display, your directory listing will appear somewhat different from the one shown above. The same diskette directory, if listed on your system, would look more like this:

```
APPLE0:
SYSTEM.PASCAL      36   4-May-79    6  512  Datafile
SYSTEM.MISCINFO     1   4-May-79   42  512  Datafile
SYSTEM.COMPILER    71  30-May-79   43  512  Codefile
SYSTEM.EDITOR      45  29-Jan-79  114  512  Codefile
SYSTEM.FILER       28  24-May-79  159  512  Codefile
SYSTEM.LIBRARY     36  22-Jun-79  187  512  Datafile
SYSTEM.CHARSET      2  14-Jun-79  223  512  Datafile
SYSTEM.SYNTAX      14  18-Apr-79  225  512  Textfile
< UNUSED >          4             239
TUNAFISH.TEXT       4   8-Jul-79  243  512  Textfile
< UNUSED >          4             247
SYSTEM.WRK.TEXT     4  17-Jul-79  251  512  Textfile
SYSTEM.WRK.CODE     2  17-Jul-79  255  512  Codefile
DRCTRY.TEXT         4  18-Jul-79  257  388  Textfile
< UNUSED >         19             261
12/12 files<listed/in-dir>, 253 blocks used, 27 unused, 19 in largest
```

The extra column of numbers gives the number of bytes used in the last block of each file. This number is almost always 512, the maximum number of bytes per block.

EXAMPLE:

Here is an E(xtended-directory-list transaction that lists a partial directory by using a wildcard in the filename specification.

Prompt: DIR LISTING OF WHAT VOL ?   or   DIR LISTING OF ?

Response: #4:S=R

Typing the response above might generate the following display:

```
APPLE0:
SYSTEM.COMPILER   71  30-MAY-79   43  CODE
SYSTEM.EDITOR     45  29-JAN-79  114  CODE
SYSTEM.FILER      28  24-MAY-79  159  CODE
< UNUSED >        93             187
3/12 FILES, 93 UNUSED, 93 IN LARGEST
```

A partial listing of a directory assumes that the last file listed is the last file on the diskette, and uses that assumption in calculating the number of unused blocks remaining on the diskette beyond the last listed file. This faulty assumption usually gives an incorrect number of unused blocks, and an incorrect size for the largest unused area. This is only a problem on partial listings; complete listings give the correct numbers.

DISK UPKEEP COMMANDS

B(ad Blocks

Scans the specified disk volume and detects bad blocks, by comparing the recorded checksum for each block with the actual information stored in the block.

This command requires that you type a volume name or number. The specified disk volume must be on-line (currently available to the system). If the disk drive or diskette is not there, the message

```
NO SUCH VOL ON-LINE <SOURCE>
```

will appear. You can just ignore the last word.

EXAMPLE:

```
Prompt:    BAD BLOCK SCAN OF WHAT VOL ?   or   BAD BLOCK SCAN OF ?

Response:  APPLE0:

Prompt:    SCAN FOR 280 BLOCKS ? (Y/N)
```

In response you will normally type Y for "Yes", telling the Filer you want to scan the entire diskette. If you wish to check only a smaller portion of the disk (a very unusual case), type N and you will be asked to type the number of blocks you want the Filer to scan.

Note: The Apple Pascal System uses only 280-block diskettes, so you can always answer the question

```
SCAN FOR 280 BLOCKS ? (Y/N)
```

with a Y . Each diskette's directory contains the number of blocks on that diskette. If your system should ever ask

```
SCAN FOR 1134 BLOCKS ? (Y/N)
```

(or any number other than 280), the diskette's directory is probably damaged.

When you type Y , the system then checks each block on the indicated diskette volume for errors, and lists the block number of each bad block. The message you will see at least 99% of the time is this:

```
SCAN FOR 280 BLOCKS ? (Y/N) Y
0 BAD BLOCKS
```

Very rarely, however, the disk drive will buzz and clatter, and you may see a message similar to this:

```
SCAN FOR 280 BLOCKS ? (Y/N) Y
BLOCK 23 IS BAD
BLOCK 24 IS BAD
BLOCK 25 IS BAD
3 BAD BLOCKS
FILE(S) ENDANGERED:
THISFILE.TEXT       18    24
THATFILE.CODE       25    29
```

The last line tells you that the three bad blocks are contained partly in the file THISFILE.TEXT , which is stored in blocks 18 through 24 , and partly in the file THATFILE.CODE , which occupies blocks 25 through 29 . Blocks reported as "bad" can often be fixed, using the X(amine command. Those which cannot be fixed can be "reserved" to avoid their use. See the X(amine command, which follows this discussion, for a more complete example of this process.

X(amine

Attempts to "fix" suspected bad blocks on a diskette (bad blocks are found by using the B(ad-blocks command). The eXamine command is invoked by typing the letter X .

This command requires that you type a disk volume name or number. The specified disk volume must be on-line (currently available to the system).

EXAMPLE:

Suppose you have just done a B(ad-blks scan of the diskette named MYDISK:, and the Filer has given you the following message:

```
SCAN FOR 280 BLOCKS ? (Y/N) Y
BLOCK 23 IS BAD
BLOCK 24 IS BAD
BLOCK 25 IS BAD
3 BAD BLOCKS
FILE(S) ENDANGERED:
THISFILE.TEXT       18    24
THATFILE.CODE       25    29
```

Now you have typed X to initiate the eXamine command.

Prompt: `EXAMINE BLOCKS ON WHAT VOLUME ?` or `EXAMINE BLOCKS ON ?`

Response: `MYDISK:`

Prompt: `BLOCK-RANGE ?`

At this point, you should have just done a bad block scan (using the B(ad-blks command), and should enter the block number returned by the bad block scan. If more than one bad block was reported, type the number of the first bad block, followed by a minus sign, followed by the number of the last bad block.

Response: 23-25

If any files are stored on the area of the diskette occupied by the blocks you are about to eXamine, you will be told the name of each such file and its beginning and ending block numbers:

```
Prompt:     FILE(S) ENDANGERED:
            THISFILE.TEXT        18    24
            THATFILE.CODE        25    29
            FIX THEM ?
```

Note: The files shown are endangered merely by their containing bad blocks, NOT by the eXamine process. Also, the question FIX THEM ? refers to the specified bad blocks, not to the files.

An N response to this prompt returns you to the outer level of the Filer. If you type a Y in response to the above prompt, you will cause the Filer to examine the blocks in the range you specified. The Filer will then usually return a message like this:

```
BLOCK 23 MAY BE OK
BLOCK 24 MAY BE OK
BLOCK 25 MAY BE OK
```

in which case the bad blocks have probably been fixed. Occasionally, however, the Filer may return a message like this:

```
BLOCK 23 MAY BE OK
BLOCK 24 IS BAD
BLOCK 25 IS BAD
FILE(S) ENDANGERED:
THISFILE.TEXT        18    24
THATFILE.CODE        25    29
MARK BAD BLOCKS  ? (FILES WILL BE REMOVED !) (Y/N)
```

in which case the Filer is offering you the option of marking the block(s) which it could not fix. If you type a Y response to this prompt, the Filer first removes all files containing those bad blocks that could not be fixed. It then creates a special file on the diskette, named BAD , which exactly covers the bad blocks (or more than one such file, if the bad blocks are not contiguous). This message then appears:

```
BAD BLOCKS MARKED
```

and you are returned to the outer Filer level. On the diskette, there is now a new directory entry saying

```
BAD.00024.BAD
```

Blocks in a file marked .BAD will not be used to store any of your files, and will not be shifted during a K(runch. These dangerous areas of your diskette are thus rendered effectively harmless. It is a good idea to do a bad-blocks scan of each new diskette, at the time you first create the formatted, zeroed diskette. Any bad spots on the diskette which are discovered at that time can be safely and easily marked, saving you much trouble in the future.

WARNING: A block which has been "fixed" may still contain useless garbage. The message MAY BE OK should be translated as "is probably physically ok". Fixing a block means that the information stored in the block is read into the computer, is stored again at the same spot on the diskette, and is then read again. If the same information is read from the block both times, that spot on the diskette is probably not physically damaged (some kinds of damage cause inconsistent recordings). In that event, the message MAY BE OK is given. However, if the two readings are different, the block is declared bad and may be marked as such to protect you from using that spot on your diskette.

The most common cause of reported bad blocks on a diskette is actual, physical damage or other problem with the diskette's recording surface. Dirt, fingerprints, and peanut butter are common culprits. An attempt to store information in a bad block may result in the loss of that information, and may render the entire file unreadable. To guard against this kind of problem, you should always do the following:

1) Handle your diskettes very carefully, and keep them clean;

2) Do a B(ad-blocks scan of every diskette at the time you format it, when you Z(ero its directory to re-use it, and at any other times when you have suspicions about the diskette.

A less common cause of bad blocks is opening the disk drive door or otherwise disturbing the recording process while the system is trying to store information on the diskette in that drive. This will sometimes create an error in the data field of a diskette sector. Since the diskette itself is not damaged, this kind of error can be "fixed" by the X(amine command. However, the information in that block may still be faulty.

Occasionally, the address field of a diskette sector may be rendered unreadable by something you or the system does. This problem is reported as a bad block by the B(ad-blocks command, but it cannot be fixed by the X(amine command. When you attempt to read or transfer the file containing the damaged address field, the system will report I/O ERROR #64 . This problem can be corrected by reformatting the diskette (which erases everything on the diskette, so be sure to save the undamaged files, first). Or, of course, you can just mark the bad blocks to avoid using them.

MISCELLANEOUS COMMANDS

P(refix

Changes the current default Prefix volume name to the volume name specified. Thereafter, until the next time the system is booted, the system will supply this Prefix volume name whenever a file specification does not include any volume name or number.

This command requires that you type a volume name or number. An entire file specification may be entered, but only the volume name or number will be used. It is not necessary for the specified volume to be on-line (actually in the system at the moment). If you specify a disk volume, either by number or by name, the Prefix is set to the NAME of the diskette in that drive. You may specify devices other than a disk volume, such as PRINTER: .

Each time the system is booted, the Prefix volume name is set to the name of the diskette in the boot disk drive (slot 6, drive 1; the boot or "Root" volume). At any time, you can return the Prefix volume name to the boot volume name again by typing an asterisk (*) in response to the Prefix prompt.

To see which volume name is currently set as the default, you may respond to the Prefix prompt by typing a colon (:) alone.

Using the C(hange command to change the name of the Prefix diskette also changes the default Prefix name supplied by the system.

On a one-drive system, being able to set the default volume name can save you much typing, if you are having to switch back and forth between two diskettes. One of the diskettes you use most will usually be the boot diskette, which can always be specified by * . If the Prefix is set to the name of the other diskette you use often, you can eliminate the volume from all file specifications typed while that diskette is in the drive.

On a two-drive system, you can use the Prefix command to make the diskette in disk drive volume #5: (slot 6, drive 2) the Prefix default. This saves typing, because the diskettes in the two drives can then be specified with the simple names : and * .

D(ate

This is the first command you should use every day; it ranks only after brushing your teeth as a necessity. The command tells you what date the system thinks it is, and allows you to correct any mistaken impression the system might have.

```
Prompt:  DATE SET: <1..31>-<JAN..DEC>-<00..99>
         TODAY IS 19-AUG-79
         NEW DATE ?
```

You may enter the correct date in the format shown after TODAY IS . For example, if today is the 13th of November, 1981, you might type

```
Response: 13-NOV-81
```

After pressing the RETURN key, the new date is displayed. Typing only a RETURN does not affect the current date. The hyphens are delimiters for the day, month and year fields, and it is possible to affect only one or two of these fields. For example, the year could be changed by typing --80 , the month by typing -SEP , etc. The entire month-name can be entered, but will be truncated to three letters by the Filer. Slash (/) is also acceptable as a delimiter.

In most cases, you will need to type only a single number, which will set the new day. For example, if yesterday was the 19th of August, you would simply type D20 and press the RETURN key. This command and response would change the date to the 20th of August (typing D initiates the date-setting routine, typing 20 sets the new date, and pressing the RETURN key terminates your entry). The day-month-year order is inviolate, however.

This date will be associated with any files saved during the current session and will be the date displayed for those files when the directory is listed. This information can be very useful when you are trying to remember which is your latest version of a file.

Q(uit

Leaves the Filer, returning you to the outermost Command level of the operating system. Remember to have your boot diskette in the boot drive before you issue this command.

FILER COMMAND SUMMARY

FILE SPECIFICATION

#5: or MYDISK:	Typical volume specification. See Table 3A for device volume names and numbers.
#5:MYFILE.TEXT or MYDISK:MYFILE.TEXT	Typical file specification. Unless otherwise noted, Filer commands require complete file specifications, including the suffix.
MYFILE.TEXT[13]	Typical filename with file [size] specifier. Used with M(ake and T(ransfer commands.
*	Specifies the boot diskette volume name.
:	Specifies the Prefix volume name.
Volume name with no filename	Specifies the entire named diskette.
Filename with no volume name	Specifies the named file on the Prefix volume.
=	Wildcard used in specifying a subset of filenames to be acted on. For example, BR=XT specifies all filenames beginning with BR and ending with XT .
?	Same as = except Filer requests verification before acting on each filename. Example: BR?XT
$	In T(ransfer, specifies a destination filename which is the same as the source filename.
,	Separates any number of Filer command response fields. Some commands use response fields in pairs.
CTRL-K	Produces the left bracket: [
SHIFT-M	Produces the right bracket:]

SYSTEM COMMANDS

CTRL-A	Shows the other 40-character "page" of the system's 80-character display, until the next CTRL-A .
CTRL-Z	Display scrolls right and left to follow the cursor.
CTRL-S	Stops any on-going process until the next CTRL-S .
CTRL-F	Flushes program output (does not send it to the screen or printer) until the next CTRL-F .

GENERAL FILE-MOVING COMMAND

T(ransfer: Transfers information from the first specified volume or file to the second specified volume or file. Destination file uses the largest unused diskette area or the first unused area of specified [size]. Used to move or save diskette files, copy entire diskettes, or send files to a printer or other device.

GENERAL DISKFILE COMMANDS

M(ake: Creates a diskette directory entry with the specified filename and [size]. Produces a "dummy" file on the diskette.

C(hange: Renames the specified diskette or diskette file to the second specified name. If second specification is a filename, it need not include the volume.

R(emove: Removes the specified file from a diskette's directory.

K(runch: Packs the files on the specified diskette so that unused portions of the diskette are combined into one area at the end (or other specified area).

Z(ero: Renames and erases the directory of the specified diskette.

WORKFILE COMMANDS

G(et: Designates a specified diskette file as the next workfile (no suffix needed: .TEXT and .CODE are supplied automatically). The next E(dit, C(ompile or R(un will use this file.

S(ave: Saves all versions of the workfile SYSTEM.WRK under the specified filename (do not specify a suffix: .TEXT and .CODE are supplied automatically).

N(ew: Clears the workfile, removing all SYSTEM.WRK files from the boot diskette.

W(hat: Tells the name and state (saved or not) of the workfile.

INFORMATION COMMANDS

V(olumes: Shows the devices and diskettes currently in the system, by volume number and by volume name.

L(ist-directory: Shows what files are on the specified diskette. If desired, list is sent to a second specified file or device.

E(xtended-directory-list: Shows what files are on specified diskette, with extra information about the files and unused portions. If desired, list is sent to a second specified file or device.

DISK UPKEEP COMMANDS

B(ad-blocks: Tests all 280 blocks on the specified diskette to see that information has been recorded consistently. Any bad blocks are reported. Use X(amine to fix bad blocks.

X(amine: Attempts to fix the specified diskette blocks, previously reported bad by the B(ad-blocks command. Allows you to mark blocks that can't be fixed, to prevent using those blocks.

MISCELLANEOUS COMMANDS

P(refix: Changes the current default volume name to the volume name specified. Response of : shows current Prefix volume name.

D(ate: Lets you specify a new current date for the system. Type one number to change the day, only.

Q(uit: Leaves the Filer and returns to the outermost Command level.

CHAPTER 4

THE EDITOR

72 INTRODUCTION
72 Diskfiles Needed
75 A "Window" into the File
76 The Cursor
76 The Prompt Line
76 Notation
77 A BRIEF SCENARIO
77 Clearing the Workfile
78 Starting a New File
78 Updating the Workfile
79 Saving the Workfile
79   One-Drive Method
80   Multi-Drive Method
81 Re-Editing an Old File
81   One-Drive Method
82   Multi-Drive Method
83 A LITTLE MORE DETAIL
83 Entering the Editor
84 Workfiles
85 Some Hidden Characters
85 Moving the Cursor
87 Using I(nsert mode
88 Using D(elete mode
88 Leaving the Editor
89 THE EDITOR COMMANDS
90 General Information
90   The Cursor
91   The Screen
91   Repeat-factors
91   The Set Direction
92   Cursor Moves
92 Moving Commands
92   J(ump
93   P(age
93   F(ind
94   Set Direction
94   Repeat-factor
94   L(iteral or T(oken Search
95   Target String and Delimiters
95   ESC Option
95   Same-string Option

- 97 Text Changing Commands
  - 97 I(nsert
    - 98 Text Formats
    - 98 With A(uto-indent TRUE, F(illing FALSE
    - 99 With A(uto-indent FALSE, F(illing TRUE
    - 100 With A(uto-indent TRUE, F(illing TRUE
    - 101 With A(uto-indent FALSE, F(illing FALSE
  - 101 D(elete
  - 103 Z(ap
  - 104 C(opy
    - 104 From a Diskette F(ile
    - 105 From the Copy B(uffer
  - 107 X(change
  - 108 R(eplace
    - 108 Set Direction
    - 109 Repeat-factor
    - 109 L(iteral or T(oken Search
    - 109 V(erify Option
    - 110 Strings
    - 110 String Delimiters
    - 111 Same-string Option
- 113 Formatting Commands
  - 113 A(djust
  - 114 M(argin
- 116 Miscellaneous Commands
  - 116 S(et
    - 116 M(arker
    - 118 E(nvironment
      - 119 The Environment Options:
        - 119 A(uto-indent
        - 119 F(illing
        - 120 L(eft Margin
        - 120 R(ight Margin
        - 120 P(aragraph Margin
        - 120 C(ommand Character
        - 121 T(oken Default
  - 122 V(erify
  - 122 Q(uit
    - 122 U(pdate the Workfile
    - 122 E(xit without updating
    - 123 R(eturn to the Editor
    - 123 W(rite to any Disk File
    - 124 S(ave to original Disk File
- 125 EDITOR COMMAND SUMMARY
- 125 Screen Commands
- 125 Special Characters
- 125 Cursor Moves
- 125 Repeat-Factor
- 125 Set Direction
- 125 Moving Commands
- 126 Text Changing Commands
- 126 Formatting Commands
- 126 Miscellaneous Commands

INTRODUCTION

DISKFILES NEEDED

The following diskfiles allow you to edit programs and text:

SYSTEM.EDITOR	(any diskette, any drive; required)
Textfile to be Edited	(any diskette, any drive; optional; default is boot diskette's workfile SYSTEM.WRK.TEXT, any drive)

The Apple Pascal system will retrieve the workfile from the boot diskette, and store the workfile onto the boot diskette, no matter which disk drive the boot diskette is in. However, since other files on the boot diskette must be found in the boot drive, it is recommended that you keep your boot diskette in the boot drive while editing.

In addition, you may wish to have some of the following diskfiles available to the system, if needed for your purposes:

SYSTEM.COMPILER	(any diskette, any drive; optional; used if you Run or Compile your text after Editing)
SYSTEM.LINKER	(any diskette, any drive; optional; used if any external routines must be Linked into your program) (no Link needed to Use intrinsic Units)
SYSTEM.LIBRARY	(boot diskette, boot drive; optional; used if Run calls the Linker, or if your program needs Long Integers, does file I/O, or USES Units)
SYSTEM.ASSEMBLER 6500.OPCODES 6500.ERRORS	(any diskette, any drive; optional; used if you Assemble your text after Editing)

The file SYSTEM.EDITOR is normally found on diskette APPLE1: and also on diskette APPLE0: . One of those diskettes is your boot diskette, so be sure that diskette is in one of your disk drives (preferably the boot drive) when you select the E(dit option.

One-drive systems boot initially with APPLE1: in the drive. If you are just Editing, Filing, and eXecuting programs that are already Compiled, you can continue to use APPLE1: as your boot diskette in a one-drive system. If you are developing a program, you will want to use the Edit-Run-Edit-Run cycle, which requires the Compiler in order to Run your newly-edited program. Since the file SYSTEM.COMPILER is not on diskette APPLE1: , one-drive users should put APPLE0: (which does contain SYSTEM.COMPILER) in the drive and press the RESET key to re-boot. Thereafter, APPLE0: is your boot diskette.

In general, one-drive users will follow this procedure when Editing:

1. If you will wish to R(un a program you are editing, you must use APPLE0: as your boot diskette. With your boot diskette (APPLE1: or APPLE0:) in the drive, enter the Filer.

2. T(ransfer onto your boot diskette a copy of the textfile you wish to Edit. Start the T(ransfer with the source diskette in the drive, and wait until prompted before putting the destination diskette (your boot diskette) into the drive.

3. With your boot diskette in the drive, G(et the textfile you have just T(ransferred. Then Q(uit the Filer, and enter the Editor. The file designated by G(et is automatically read into the Editor.

4. Edit the file. Q(uit, U(pdate the workfile, and re-enter the Editor from time to time. On each re-entry, the updated workfile (SYSTEM.WRK.TEXT) is automatically read back into the Editor. When you are through editing, Q(uit and U(pdate the workfile one last time, but do not re-enter the Editor.

5. If you are editing a program, you can R(un the program now to check its operation and also to generate a code version (SYSTEM.WRK.CODE) of your latest workfile. Repeat steps 4 and 5 until the program runs as it should.

6. Enter the Filer and use the S(ave command to rename the workfile on the boot diskette. Then T(ransfer the S(aved file or files, one at a time, onto any other diskette. Start each T(ransfer with the boot diskette in the drive, and wait until prompted before putting the destination diskette in the drive.

7. You may also wish to R(emove the S(aved files from your boot diskette at this time, to leave more room on that diskette for future editing jobs.

8. Before you Exit the Filer, put your boot diskette back in the drive.

Two-drive systems also boot with APPLE1: in the boot drive. If you place APPLE2: in the other drive, your system can use SYSTEM.COMPILER from that diskette when you want to R(un a program you are editing.

Another possibility is to press RESET with APPLE0: in the boot drive. Diskette APPLE0: (which contains SYSTEM.COMPILER) then becomes your boot diskette and you will not have to put APPLE2: in the non-boot drive to R(un a program you are editing.

On two-drive systems, "non-boot drive" means drive volume #5: (the "boot drive" is volume #4:). On systems with three or more drives, "non-boot drive" means any drive except volume #4:. Systems with three or more drives can leave APPLE1: and APPLE2: in drive volumes #4: and #5: throughout the Edit-Run-Edit-Run cycle.

In general, two-drive users will follow this procedure when Editing:

1. With your boot diskette (APPLE1: or APPLE0:) in the boot drive, put in the other drive the diskette that has the file you wish to Edit.

2. Enter the Filer, and G(et the textfile you wish to Edit. Then Q(uit the Filer, and enter the Editor. The file designated by G(et is automatically read into the Editor.

3. Edit the file. Q(uit, U(pdate the workfile, and re-enter the Editor from time to time. On each re-entry, the updated workfile (SYSTEM.WRK.TEXT) is automatically read back into the Editor. When you are through editing, Q(uit and U(pdate the workfile one last time, but do not re-enter the Editor.

4. If you are editing a program, you can R(un the program now to check its operation and also to generate a code version (SYSTEM.WRK.CODE) of your latest workfile. If your boot diskette is APPLE1: , you should put APPLE2: in the non-boot drive before attempting to R(un your program. This is not necessary if you are using APPLE0: as your boot diskette. Repeat steps 3 and 4 until the program runs as it should.

5. Enter the Filer, and S(ave the workfile onto a diskette that you have put in the non-boot drive.

If you are only editing text, you may wish to remove all unnecessary files from a copy of APPLE1: , in order to leave room for large text files on your boot diskette. The following example shows a directory list of files on a possible text-editing-only diskette named EDIT1:

```
EDIT1:
SYSTEM.APPLE        32  26-JUL-79       6  DATA
SYSTEM.PASCAL       36   4-MAY-79      38  DATA
SYSTEM.MISCINFO      1   4-MAY-79      74  DATA
SYSTEM.EDITOR       45  29-JAN-79      75  CODE
SYSTEM.FILER        28  24-MAY-79     120  CODE
< UNUSED >         132                148
```

When you are handling large text files, the amount of unused space on the boot diskette is important. During the course of editing, the file being worked on is usually stored again and again in the workfile on the boot diskette. To be safe, the contiguous unused space available for storing the workfile should be at least THREE TIMES the size of the largest workfile you will store. Since text files can be as large as 38 blocks, an unused area of about 114 blocks would be safe. Of course, for working on smaller programs and text much less room is needed.

Note that text files always use diskette space in two-block increments.

One-drive systems must also have room on the boot diskette for the original copy of the file being edited, in addition to room needed for the workfile. If space is a real problem, you could avoid U(pdating the workfile by always W(riting to the name of the original copy on the boot diskette, instead. That way, the file SYSTEM.WRK.TEXT would not be created. The old version of your original copy will be removed automatically after each time you W(rite the newest version to that name.

Two-drive systems can keep in another drive the original copy of the file being edited, so that only the workfile SYSTEM.WRK.TEXT appears on the boot diskette in the boot drive.

A "WINDOW" INTO THE FILE

The Screen-Oriented Editor is specifically designed for use with video displays such as the Apple's TV or monitor. On entering any file, the Editor displays the start of the file on the second line of the screen. If the file is too long for the screen, only the first portion is displayed. This is the concept of a "window". The whole file is there and is accessible by Editor commands, but only a portion of the file can be seen through the "window" of the screen. When any Editor command would take you to a position in the file which is not displayed, the "window" is moved to show that portion of the file.

The Apple Pascal Editor uses a text window that is 80 characters wide. On the Apple's TV or monitor, only the leftmost 40 characters of the window are normally displayed. To see the rightmost 40 characters of the window at any time, just press CTRL-A. Frequently, the right half of the window is just black, as there is no text to display there. Pressing CTRL-A again shifts you back to the left half of the window.

You can also make the display "scroll" to the right and left, by pressing CTRL-Z. In the CTRL-Z mode, the display "follows" the cursor everywhere it goes. As the cursor moves, the display is automatically adjusted to show the text surrounding the cursor. CTRL-Z is cancelled by CTRL-A and by many other commands.

Most programs will not require you to write beyond the leftmost 40 characters. For many other text applications, you can adjust the display's right margin to column 39 (see the E(nvironment command), so that the text will be confined to the leftmost 40 characters of the window.

THE CURSOR

The cursor marks a position in the file and can be moved to any position occupied by text. The window shows a portion of the file near the cursor. To see another portion of the file, move the cursor. Action usually takes place in the vicinity of the cursor.

If the text being displayed is more than 40 characters wide, the cursor will disappear when you move it right, beyond the leftmost 40 characters. To see the cursor and the text around it, you can press CTRL-A (which shows you the other half of the 80-character Apple Pascal window), or you can invoke the "Auto-Follow" option by pressing CTRL-Z . After CTRL-Z, the Apple's 40-character display will automatically scroll right and left to follow the cursor wherever it goes, so the cursor never disappears from view. CTRL-A and many other commands cancel the action of CTRL-Z.

There are a number of commands available to you. Some of the commands permit additions, changes or deletions of such length that the screen cannot hold the whole portion of the text that has been changed. In those cases, the screen shows the portion of the file where the cursor ended up after the change. In no case is it necessary for you to operate on portions of the text not seen on the screen, but in some cases it is optional.

THE PROMPT LINE

The Editor's prompt line lets you know, first of all, that you are in the Editor rather than in some other part of the system. The Apple Pascal operating system is complex enough that you need these signposts to remember where you are in the system. Secondly, the prompt line reminds you of some of the commands you can use. Remember that on the Apple's 40-character screen display, you will sometimes see only the leftmost 40 characters of the prompt line. Use CTRL-A to see the rest of the line.

Here is the complete Editor prompt line:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

The letters and numbers in square brackets which will appear on your screen at the end of the prompt line are just the version number for this portion of the program.

NOTATION

The notation used in this chapter is sometimes borrowed from the notation used in the Editor prompt lines. In the Editor prompt lines, a word enclosed between angle-brackets < like this > tells you that a particular key is to be pressed. For example, <RET> means that the RETURN key should pressed at that point, and <ESC> means to press the ESC key. Either lower-case or upper-case characters may be used when typing Editor commands.

A BRIEF SCENARIO

This scenario will give you a quick idea of what is involved in using the Editor, with little or no attempt to explain the terms and concepts used. Following the scenario is a more detailed discussion of the same concepts, in the section called A LITTLE MORE DETAIL. Following that section there is a full discussion of all Editor commands.

CLEARING THE WORKFILE

From Command level, with your boot diskette (APPLE0: or APPLE1:) in the boot disk drive (volume #4:), type F for F(ile . The following prompt line appears:

```
FILER: G(ET, S(AVE, W(HAT, N(EW, L(DIR, R(EM, C(HNG, T(RANS, D(ATE, Q(UIT
or
FILER: G, S, N, L, R, C, T, D, Q [C.2]
```

Type N for N(ew . If this message appears:

```
          WORKFILE CLEARED
```

then you can simply type Q to Q(uit the Filer, and proceed to the next section. However, if this message appears:

```
          THROW AWAY CURRENT WORKFILE ?
```

you may be about to lose someone's valuable workfile. Unless you know for sure that all files beginning with SYSTEM.WRK on the boot diskette are dispensable (these files constitute the workfile), you should respond to the above question by typing an N for "No". Now type S for S(ave , and the system will ask you

```
          SAVE AS WHAT FILE ?     or     SAVE AS ?
```

Respond by typing any valid filename (without any .TEXT or .CODE suffix). For example, you might type

```
          OLD.WRK
```

The system obediently renames SYSTEM.WRK.TEXT as OLD.WRK.TEXT , and renames SYSTEM.WRK.CODE (if there is one) as OLD.WRK.CODE . When this is done, again type N for N(ew , and this time you will see

```
          WORKFILE CLEARED
```

Type Q for Q(uit to return to the system's outermost Command level.

STARTING A NEW FILE

Now that the workfile has been cleared, type E for E(dit from the Command level. Soon, this prompt appears:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
:
```

Press the RETURN key to start a new file, and this prompt appears:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

You can now type I to enter I(nsert mode, and then proceed to type your program or text. When each portion of the insertion is complete, you press CTRL-C to accept that portion of the insertion and terminate I(nsert mode. If you wish to add more text, type I again to re-enter I(nsert mode. Further typing will insert text at the cursor position, until you terminate the latest insertion by pressing CTRL-C.

If you wish to remove unwanted text after CTRL-C has terminated an insertion, move the cursor to any appropriate part of the text (using the arrow-keys), and then type D to enter D(elete mode. In D(elete mode, moving the cursor erases the characters moved over. Terminate each deletion by pressing CTRL-C.

For example, you might type I for I(nsert, and then type

```
PROGRAM EXAMP;
BEGIN
  WRITE('AN APPLE A DAY')
END.
```

Terminate this insertion by pressing CTRL-C.

UPDATING THE WORKFILE

Finally, when the text is the way you want it for now, type Q for Q(uit and then type U for U(pdate. The system stores your file in the workfile, a file on the boot diskette called SYSTEM.WRK.TEXT , and you are once again at the outermost Command level.

If your file was a Pascal program (as the example is), you may now type R for R(un, and the system will automatically attempt to compile and run the workfile, storing the compiled version of your program (if compilation is successful) as SYSTEM.WRK.CODE . If APPLE1: is your boot diskette, APPLE2: must be in another drive during compilation. APPLE2: is not necessary if APPLE0: is your boot diskette.

If you wish to change your file for any reason, simply type E for E(dit again. Now a workfile is present, and the system automatically reads the workfile into the computer, ready for more Editing.

SAVING THE WORKFILE

One-Drive Method

On one-drive systems, you can only S(ave one version of the workfile (usually .TEXT) onto another diskette. To save more than one workfile version (usually .TEXT and .CODE), you must first S(ave all versions onto the boot diskette, and then T(ransfer each version to the other diskette. Then you can R(emove the S(aved files from the boot diskette. Here is how it might be done:

When the U(pdated workfile contains your finished product, or when you need to start a new file for another project, type F from Command level to enter the Filer. From the Filer, type S for S(ave and you are prompted:

```
SAVE AS WHAT FILE ?    or    SAVE AS ?
```

You should respond by typing a valid boot diskette file specification (without any .TEXT or .CODE suffix). The system then renames all versions of the workfile to the filename which you have specified. For example, if APPLE0: is your boot diskette, you might respond by typing

```
APPLE0:PROGRAM1
```

The system obediently renames SYSTEM.WRK.TEXT as PROGRAM1.TEXT, and SYSTEM.WRK.CODE (if it exists) as PROGRAM1.CODE , on the boot diskette. This step makes the former workfile safe from being accidentally erased by a N(ew command, and tells the system that the workfile is gone.

Now type T for T(ransfer. The system prompts

```
TRANSFER WHAT FILE ?      or      TRANSFER ?
```

You should respond by typing the complete file specification (including the suffix, this time) for one version of your saved file. In the example, you might type

```
APPLE0:PROGRAM1.TEXT
```

When you press the RETURN key, the system asks

```
TO WHERE ?
```

Now type the complete file specification for the destination file. If you wish to save your PROGRAM1 files on diskette MYDISK: , for example, you would type

```
MYDISK:PROGRAM1.TEXT
```

The disk drive whirrs, and soon this message appears:

```
PUT IN MYDISK:
TYPE <SPACE> TO CONTINUE
```

Follow the directions, putting diskette MYDISK: in the disk drive and pressing the Apple's spacebar. When PROGRAM1.TEXT has been successfully transferred to MYDISK: , you can put APPLE0: back in the disk drive.

Now, repeat the T(ransfer command, this time saving the file PROGRAM1.CODE (if it exists) onto MYDISK: . When that transfer is complete, again put APPLE0: back in the disk drive.

To prevent your boot diskette from becoming cluttered up with files that you have already saved elsewhere, you may wish to remove the PROGRAM1 files from APPLE0: at this time. Type R for R(emove, and when the Filer prompts

```
REMOVE WHAT FILE ?      or      REMOVE ?
```

type the complete file specification (including the .TEXT or .CODE suffix) of one file that you wish removed from the boot diskette. For example, you might respond by typing

```
APPLE0:PROGRAM1.TEXT
```

The Filer soon says

```
APPLE0:PROGRAM1.TEXT            --> REMOVED
UPDATE DIRECTORY ?
```

This gives you a last chance to avoid removing the specified file by typing an N response. If you type a response of Y , the file PROGRAM1.TEXT is removed from APPLE0:'s directory, and the system forgets that file's existence. You can repeat the R(emove command as often as you wish, of course, until all unnecessary files have been removed.

Multi-Drive Method

On multiple drive systems, you can S(ave all the versions of the workfile (usually .TEXT and .CODE) directly onto another diskette, using a filename of your choice. Then N(ew erases the workfile from the boot diskette. This is the process:

When the U(pdated workfile contains your finished product, or when you need to start a new file for another project, type F from Command level to enter the Filer. From the Filer, type S for S(ave and you are prompted:

```
SAVE AS WHAT FILE ?     or     SAVE AS ?
```

When you respond by typing any valid disk file specification (without any .TEXT or .CODE suffix), the system saves all versions of the workfile under the filename which you have specified. For example, if you respond by typing

```
MYDISK:PROGRAM1
```

the system saves SYSTEM.WRK.TEXT as PROGRAM1.TEXT, and SYSTEM.WRK.CODE (if it exists) as PROGRAM1.CODE , on diskette MYDISK:

You can now type the Filer command N(ew , which erases all versions of the workfile on the boot diskette, and the creation or editing process can begin again.

RE-EDITING AN OLD FILE

One-Drive Method

On one-drive systems, you must first use the Filer to T(ransfer onto your boot diskette the file you want to edit. Only then can you use the Filer to G(et that file.

From Command level, with your boot diskette (APPLE0: or APPLE1:) in the disk drive, type F to enter the Filer. Now put into the disk drive the diskette containing the file you wish to edit. Type T for T(ransfer, and when you see the message

```
TRANSFER WHAT FILE ?      or      TRANSFER ?
```

respond by typing the complete file specification (including the .TEXT suffix) for the textfile you want to edit. For example, to re-edit the file saved on MYDISK: in the previous section, you might type

```
MYDISK:PROGRAM1.TEXT
```

When you press the RETURN key, the disk whirrs and this message appears:

```
TO WHERE ?
```

At this point you should type the complete file specification (again including the .TEXT suffix) for the file as it will appear on your boot diskette. For example, if you are using APPLE0: as your boot diskette, you might type

```
APPLE0:PROGRAM1.TEXT
```

After some more disk-whirring, you will be prompted

```
PUT IN APPLE0:
TYPE <SPACE> TO CONTINUE
```

Put APPLE0: in the disk drive, press the spacebar, and a copy of PROGRAM1.TEXT is saved on APPLE0: .

Now that the file you want to edit is on the boot diskette, you can type G to G(et that file. When you see the prompt

```
GET WHAT FILE ?      or      GET ?
```

type the file specification (without the .TEXT suffix, this time) for the boot diskette copy of the file you want to edit. For the example we have been showing, you would respond to the prompt by typing

```
APPLE0:PROGRAM1
```

When the Filer has cleared away any previous workfile (as the command N(ew did), it marks the specified file as the next workfile. Since only the .TEXT version of PROGRAM1 exists on APPLE0: , you will soon see the message

```
TEXT FILE LOADED
```

Now type Q to Q(uit the Filer, and from Command level type E for E(dit. The file designated by G(et is automatically read into the Editor, ready for work.

Multi-Drive Method

On multiple-drive systems, you can use the Filer to G(et the file you want to edit, directly from any diskette in any available drive.

Put your boot diskette (APPLE1: or APPLE0:) in the boot drive and put into another drive the diskette containing the old textfile you wish to edit. From Command level, type F to enter the Filer, and then type G for G(et . When you see the prompt

```
GET WHAT FILE ?      or      GET ?
```

type the file specification for the file you want to edit. For example, to re-edit the file saved on MYDISK: in the example earlier in this section, you might respond to the prompt by typing

```
MYDISK:PROGRAM1
```

When the Filer has cleared away any previous workfile (as the command N(ew did), it marks the specified file as the next workfile. Since both .TEXT and .CODE versions of PROGRAM1 exist on MYDISK: , you will soon see the message

```
TEXT & CODE FILE LOADED
```

This does not mean that PROGRAM1 has been transferred to your boot diskette or into memory; you must continue to leave MYDISK: in its drive. Now type Q to Q(uit the Filer, and from Command level type E for E(dit. The file designated by G(et is automatically read into the Editor, ready for work. THEN you may take MYDISK: out of its drive, if you wish.

A LITTLE MORE DETAIL

ENTERING THE EDITOR

When the COMMAND: prompt line is on the screen, and your boot diskette (usually APPLE0: or APPLE1:) is in the boot drive (volume #4:), type an E for E(dit . If the system already has a text workfile (see WORKFILES, below), that file is automatically read into the Editor, ready for work. If the system does not have a workfile yet, or if only a code workfile exists, this prompt line appears on first entering the Editor:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
:
```

Of course, you may not be able to see the entire prompt line at once. Use CTRL-A to flip back and forth between the rightmost 40 characters of the display and the leftmost 40 characters.

There are three ways to answer this opening prompt line's question:

1. You can respond by typing the file specification of any textfile that already exists on diskette.

   On one-drive systems, you must specify a textfile on the boot diskette. On multi-drive systems, the diskette containing the specified file may be in any drive. The Editor will read the specified file into the computer, and then display the first part of the file's text on the screen, ready for Editing.

   For example, you might type

   ```
   APPLE0:PROGRAM1
   ```

   When you press the RETURN key , the file named PROGRAM1.TEXT is retrieved from diskette APPLE0: and the text of that file is displayed on the screen.

2. You can answer by pressing the RETURN key: <RET>

   This tells the system that you are starting a new file. The only thing visible on the screen after doing this is the normal EDIT prompt line:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

   A new file has been started and currently has nothing in it. Type I to begin I(nserting a program or text. No diskette version of this new file exists until you Q(uit the Editor and U(pdate the workfile or W(rite to some other disk file.

3. You can answer by pressing the ESC key and then pressing the RETURN key: <ESC-RET>

   This causes the Editor to return you to the system Command level, a useful option when you didn't mean to type E .

WORKFILES

The workfile is a diskette "scratchpad" copy of the file on which you are currently working. Each time you Q(uit the Editor and U(pdate (see LEAVING THE EDITOR, at the end of this section), the latest version of your program or text is saved in the workfile, under the name SYSTEM.WRK.TEXT on the boot diskette. This is a very convenient arrangement, as you will see.

The first convenience comes each time you type E to enter the Editor from Command level. No questions are asked if a workfile already exists (in a file named SYSTEM.WRK.TEXT on the boot diskette). The workfile is automatically read into the computer and displayed on the screen, ready for editing. If you keep the workfile U(pdated to the latest version of a job in progress, you can turn on the computer at any time, type E from Command level, and you will immediately be ready to work on your job again.

Editing using the workfile has another advantage when you are working on a program. You can Q(uit the Editor and U(pdate the workfile, and then immediately type R from Command level to compile and R(un the workfile automatically. Following a successful compilation, the compiled version of the workfile is automatically saved as SYSTEM.WRK.CODE . After the code version of your program has been executed, typing E from Command level will automatically read the text version of your program back into the Editor for more work. Thus you are almost completely spared the constant typing of filenames during program development.

Ordinarily, the workfile is created by Q(uitting the Editor and U(pdating the workfile. There is also one other way to make the system behave as if there were a workfile. From the Filer, the G(et command can be used to designate any textfile as the next workfile. The file so designated will be read into the computer when you next enter the Editor, just as if that file were on the boot diskette and named SYSTEM.WRK.TEXT . This designation is "forgotten", however, each time you boot the system.

When you have completed work on your file, you will want to enter the Filer and S(ave the latest U(pdated workfile under some other name, or else you will Q(uit the Editor by W(riting to a disk file of some other name. But during the development of a program or text file, it is very convenient to keep the latest version saved in the workfile.

To edit a different file when a workfile already exists, the workfile must first be cleared, by using the Filer's N(ew command (if you wish to start a new file) or the G(et command (if you wish to re-edit an old file). Make sure that you have S(aved the latest version of the workfile under another name, before you clear the workfile, as all versions of SYSTEM.WRK will be erased.

SOME HIDDEN CHARACTERS

The Apple II keyboard doesn't appear to have the left and right bracket symbols [and] . But they can by typed. The left bracket is produced by typing CTRL-K. The right bracket is produced by typing SHIFT-M. You might consider marking these special characters on your keyboard.

MOVING THE CURSOR

In order to edit, it is necessary to move the cursor. On the keyboard are two "arrow-keys" which move the cursor right and left. In addition to these cursor-moving keys, CTRL-O moves the cursor up one line, and CTRL-L moves the cursor down one line. Some people find it convenient to mark these last keys with an up-arrow and a down-arrow, to help them remember. You can move the cursor only when one of these prompt lines is at the top of the screen: EDIT , DELETE , or ADJUST .

If you type a number before you type a cursor move, the cursor moves that number of characters or lines in the direction indicated. Typing P moves the cursor to the next "page", a little more than a screenful away from the current cursor position. Also, notice how the spacebar and the RETURN key move the cursor. Sometimes these moves are useful.

Vertical motion of the cursor is made without regard to the text on the page. But for horizontal moves, the cursor does not like to be outside of the text of the program. For example, suppose the cursor appears after the "N" in "BEGIN" :

```
PROGRAM EXAMP;
BEGIN
  WRITE('AN APPLE A DAY')
END.
```

(Actually, the cursor is "on" the invisible RETURN character that ends every line.) If you press the right-arrow key, the cursor moves to the "W" in "WRITE" :

```
PROGRAM EXAMP;
BEGIN
  WRITE('AN APPLE A DAY')
END.
```

Similarly, pressing left-arrow key now moves the cursor back to after the "N" in "BEGIN".

If it is necessary to change "WRITE('AN APPLE A DAY')" found in the third line to "WRITE('AN ORANGE A DAY')", the cursor must first be moved to the correct spot.

For example: if the cursor is on the "P" in "PROGRAM EXAMP;", go down two lines by pressing CTRL-L twice. After the first CTRL-L the cursor is on the "B" in "BEGIN"; and after the second CTRL-L the cursor is in front of the "W" in "WRITE".

```
PROGRAM EXAMP;
BEGIN
  WRITE('AN APPLE A DAY')
END.
```

Now, using the right-arrow key, move until the cursor sits on the "A" in "APPLE".

Note that with downward and upward cursor moves (using CTRL-L and CTRL-O) the cursor may at times appear to be outside the text. In the last illustration, the cursor appears to be in the blank space before the "W" in "WRITE". As far as the Editor knows, however, the cursor is actually on the "W" in "WRITE". So do not be surprised when, on first pressing the right-arrow key, the cursor jumps to the "R" in "WRITE". In other words, when the cursor appears to be outside the text, it is conceptually on the closest character to the right or left.

Remember that the Apple's TV screen only shows the leftmost 40 characters of the system's 80-character-wide display. When the cursor disappears into the hidden portion of the Pascal display, you can follow it by pressing CTRL-A . Even easier, you can initiate "Auto-follow" mode, by pressing CTRL-Z . After CTRL-Z , Apple's 40-character screen automatically scrolls left and right to keep the cursor visible. CTRL-A and many other commands cancel CTRL-Z .

USING I(NSERT MODE

The EDIT prompt line shows the command option I(nsrt (or "Insert", if you like your words with all their vowels and without parentheses running amok inside them). To Insert an item, first move the cursor to the correct position, and then type I . You must always move the cursor to the correct position BEFORE typing I . Earlier, the cursor was moved to the "A" in "APPLE". Now, on typing I , an insertion will be made just before (just to the left of) the "A". The rest of the line starting with the "A" will be moved to the right hand side of the screen. If the insertion is lengthy, the right hand portion of the line (beginning with "A") will be moved down to allow room on the screen for more inserted text to appear. After typing I the following prompt line should appear on the screen:

```
>INSERT: TEXT [<BS> A CHAR,<DEL> A LINE] [<ETX> ACCEPTS, <ESC> ESCAPES]
```

If that prompt line did not appear at the top of the screen, you are NOT in Insert mode. You may have typed a wrong key.

If the cursor was at the "A" in "APPLE" when you typed I , the Insert prompt line appeared and the remaining portion of that line (beginning with "A") was pushed to the right hand edge of the screen. "ORANGE" may be inserted by typing those six letters. They will appear on the screen as they are typed.

There remains one more important step. The choice at the end of the prompt line indicates that <ETX> (which means pressing CTRL-C) accepts the insertion, while <ESC> (which means pressing the ESC key) rejects the insertion so that the text remains as it was before typing I .

(Portion of screen when Inserting ORANGE)

```
----------------------------------------------------------------------
PROGRAM EXAMP;
BEGIN
  WRITE('AN ORANGE                                       APPLE A DAY')
END.
----------------------------------------------------------------------
```

(Portion of screen after Insertion followed by CTRL-C)

```
----------------------------------------------------------------------
PROGRAM EXAMP;
BEGIN
  WRITE('AN ORANGEAPPLE A DAY')
END.
----------------------------------------------------------------------
```

(Portion of screen after Insertion followed by <ESC>)

```
-------------------------------------------------------------------
PROGRAM EXAMP;
BEGIN
  WRITE('AN APPLE A DAY')
END.
-------------------------------------------------------------------
```

It is legal to Insert a carriage return. This is done by doing a <RET> (that is, pressing the RETURN key) while in the Insert mode and causes the Editor to start a new line.

USING D(ELETE MODE

The Delete mode works somewhat like the Insert mode. Having inserted the word "ORANGE" into the EXAMP program and having typed CTRL-C, "APPLE" must now be deleted. Move the cursor so that it is placed directly on the first character that you wish to delete. Then type D to put the Editor into Delete mode. The following prompt line should appear.

```
>DELETE: < > <MOVING COMMANDS> [<ETX> TO DELETE, <ESC> TO ABORT]
```

Remember that <ETX> means to type CTRL-C.

Each time the right-arrow key is pressed, the character on which the cursor is sitting disappears. Pressing the left-arrow key will erase characters to the left of the first cursor position. In this example, pressing the right-arrow key five times will cause the word "APPLE" to disappear. To terminate the Deletion, you have the same choice you had with Insert. Use <ETX> (by typing CTRL-C) to make the proposed deletion permanent. Use <ESC> (by pressing the ESC key) to cancel the proposed deletion and restore the original text.

It is legal to delete a carriage return. At the end of the line, enter Delete mode and press the right-arrow key until the cursor moves to the beginning of the next line.

LEAVING THE EDITOR

When all the changes and additions have been made in your program or text, you will want to exit the Editor and "save" a copy of the modified program or text. This is done by typing Q for Q(uit, which will cause this prompting display:

```
>QUIT:
     U(PDATE THE WORKFILE AND LEAVE
     E(XIT WITHOUT UPDATING
     R(ETURN TO THE EDITOR WITHOUT UPDATING
     W(RITE TO A FILE NAME AND RETURN
     S(AVE WITH SAME NAME AND RETURN
```

The most elementary way to save a copy of your present file onto disk is to type U for U(pdate. This causes your file to be saved in the workfile on the boot diskette, under the filename SYSTEM.WRK.TEXT. With the workfile thus saved, it is possible to use the R(un command, provided of course the file is a program.

It is also possible to use the S(ave option in the Filer to save the diskette workfile under its own filename before using the Editor to modify or create another file. Remember that the Filer's N(ew command can erase the workfile SYSTEM.WRK.TEXT at any time, and that the Editor's U(pdate always stores the just-edited file under the same filename SYSTEM.WRK.TEXT. You will not want SYSTEM.WRK.TEXT to be your only copy of a file, once you are through working on it.

It is a good idea to temporarily Q(uit and U(pdate your file about every 15 minutes or so. This way, in case of accident (such as the power going out, or your mistakenly deleting an important part of your file), you are not likely to lose more than 15 minutes worth of typing.

These are sufficient commands to edit any file desired. The next section describes many more commands in the Editor which make editing even easier.

THE EDITOR COMMANDS

90 General Information
  90 The Cursor
  91 The Screen
  91 Repeat-factors
  91 The Set Direction
  92 Cursor Moves
92 Moving Commands
  92 J(ump
  93 P(age
  93 F(ind
    94 Set Direction
    94 Repeat-factor
    94 L(iteral or T(oken Search
    95 Target String and Delimiters
    95 ESC Option
    95 Same-string Option
97 Text Changing Commands
  97 I(nsert
    98 Text Formats
    98 With A(uto-indent TRUE, F(illing FALSE
    99 With A(uto-indent FALSE, F(illing TRUE
    100 With A(uto-indent TRUE, F(illing TRUE
    101 With A(uto-indent FALSE, F(illing FALSE

101 D(elete
103 Z(ap
104 C(opy
104 From a Diskette F(ile
105 From the Copy B(uffer
107 X(change
108 R(eplace
108 Set Direction
109 Repeat-factor
109 L(iteral or T(oken Search
109 V(erify Option
110 Strings
110 String Delimiters
111 Same-string Option
113 Formatting Commands
113 A(djust
114 M(argin
116 Miscellaneous Commands
116 S(et
116 M(arker
118 E(nvironment
119 The Environment Options:
119 A(uto-indent
119 F(illing
120 L(eft Margin
120 R(ight Margin
120 P(aragraph Margin
120 C(ommand Character
121 T(oken Default
122 V(erify
122 Q(uit
122 U(pdate the Workfile
122 E(xit without updating
123 R(eturn to the Editor
123 W(rite to any Disk File
124 S(ave to original Disk File

GENERAL INFORMATION

The Cursor

The "cursor" is the white rectangle that indicates your position in the file. In general, special "cursor moves" (see below) are used in the Editor to move the cursor through the text and place it just where you want the next command to have its effect. Once you have initiated a particular command, the same cursor moves may have an additional function, such as deleting text or adjusting the position of lines.

It should be pointed out that not all commands affect the character on which the cursor is actually sitting. Some commands do affect the character AT the cursor position (eXchange, for example, and Delete when used with the right-arrow key). But other commands affect the first character to the LEFT of the cursor position (for example, Delete when used with the left-arrow key). Still other commands affect the place BETWEEN the cursor position and the first character to its left (for example, Insert). A little experience, or a careful reading of the detailed command descriptions, will teach you what to expect.

THE SCREEN

The Apple Pascal system's display is always 80 characters wide, but the Apple screen normally shows only the leftmost 40 characters. If the cursor should disappear into the hidden half of the display, use CTRL-A to see the other half of the display. To initiate "Auto-follow" mode, in which the Apple screen automatically scrolls left and right to keep the cursor visible, use CTRL-Z . CTRL-A and many other commands cancel CTRL-Z .

Repeat-Factor

Most of the cursor moves, and some of the command options, allow repeat-factors. A repeat-factor is a number which is typed immediately before issuing a cursor move or command. The cursor move or the command option is then repeated for the number of times indicated by the repeat-factor. For example, typing 2 followed by CTRL-L causes the cursor move to be executed twice, moving the cursor down two lines. Cursor moves and commands which allow a repeat-factor assume the repeat-factor to be 1 if no number is typed. Typing a slash (/) before a cursor move or a command indicates an "infinite" repeat-factor, and causes the move or command to be repeated as many times as possible in the file.

The Set Direction

The first character displayed on most Editor prompt lines is a "direction indicator". On entering the Editor, the set direction is "forward". A "greater than" (>) character indicates "forward" direction. A "less than" (<) character indicates "backward" (or reverse) direction. When the EDIT or the DELETE prompt line is showing, you can type the > or the < key (with or without the SHIFT key) to change the set direction.

Typing one of these keys:			Changes the set direction to:
,	<	-	backward
.	>	+	forward

Certain commands and certain cursor moves are affected by the set direction. If the set direction is forward, then they operate forward through the file, that being the standard direction of reading English. Forward operations begin at the current cursor position and proceed toward the end of the file. Backward is the reverse direction. Backward operations begin at the current cursor position and proceed toward the beginning of the file. Where the set direction affects a command, this is specifically noted in the detailed command description.

Cursor Moves

If you type:	The cursor moves:
CTRL-L	down
CTRL-O	up
right-arrow key	right
left-arrow key	left
spacebar	in the set direction
CTRL-I or TAB key	in the set direction, to the next tab-stop (tab-stops are set every 8 spaces across the screen)
RETURN key	in the set direction, to the beginning of the next line
= (equals)	to the beginning of the last text Inserted, Found, or Replaced

Repeat-factors can be used with any of the above commands (except the "equals" command.

The Editor maintains the column position of the cursor when you use CTRL-O and CTRL-L, even when this means that the cursor appears outside the text. If the cursor appears to the right of a line of text, the Editor acts as though the cursor were immediately after the last character in the line. If the cursor appears to the left of a line of text, the Editor acts as though the cursor were on the first character in the line.

MOVING COMMANDS

J(ump

Jump mode is reached by typing J for J(ump while at the Edit level. On entering Jump mode the following prompt line appears:

```
>JUMP: B(EGINNING E(ND M(ARKER <ESC>
```

Typing B for B(eginning moves the cursor to the beginning of the file, displays the Edit prompt line and the first page of the file. Typing E for E(nd moves the cursor to the end of the file, displays the Edit prompt line and the last page of the file. Typing M for M(arker causes the Editor to display the prompt line:

```
JUMP TO WHAT MARKER?
```

If you respond by typing the name of a marker that exists in the file, when you press the RETURN key the cursor is placed at the marker position in the text. If you type a marker name that does not exist in the file, this message is given:

```
ERROR: NOT THERE.  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

and the cursor is not moved. Placing markers in the text is explained under S(et M(arker , in the Miscellaneous Commands.

P(age

Page command is executed by typing P while at the Edit level. Depending on the set direction, indicated at the beginning of the EDIT prompt line, Page command moves the cursor somewhat more than one whole screenful up or down. The cursor always moves to the start of a line. A repeat-factor may be used before this command, for moving several pages.

F(ind

Find mode is reached by typing F for F(ind while at the Edit level. On entering Find mode one of the following prompt lines appears, depending on the setting of the Environment's T(OKEN DEFault option (see S(et E(nvironment, in Miscellaneous Commands):

```
>FIND[1]: L(IT <TARGET>  =>
```

if the Environment's T(OKEN DEFault option is set to TRUE, or

```
>FIND[1]: T(OK <TARGET>  =>
```

if the Environment's T(OKEN DEFault option is set to FALSE.

Find mode searches through a file in the set direction, finds the repeat-factor-th occurrence of the specified string of characters <TARGET>, and places the cursor at the end of that string.

Set direction

The F(ind command searches for the specified occurrence of the target string beginning at the present cursor position and scanning through the text in the set direction (indicated by the arrow at the beginning of the prompt line). An occurrence of the target string will be found only if it appears in that portion of the text which lies between the cursor and the end of the file toward which the search is progressing. See the section on the set direction (in this chapter, under General Information) in order to change the set direction arrow. If the required occurrence of the target string is not found by searching through the text in the set direction, this message appears:

```
ERROR: PATTERN NOT IN THE FILE  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

Remember, however, that the search does not "wrap around". That portion of the file between the cursor and the end of the file in the direction OPPOSITE the set direction is not searched.

Repeat-factor

The repeat-factor is an integer from 0 to 9999 which may be typed just before typing the F for F(ind. It is shown on the prompt line in square brackets: [1] , for example. If a repeat-factor of n is specified, the cursor is placed after the n-th occurrence of the target string. If no repeat-factor is specified, a repeat-factor of one is used. If a repeat-factor of / is used, the cursor is placed after the last occurrence of the specified string.

L(iteral or T(oken search

The target string is treated somewhat differently, depending on whether Literal search or Token search is selected. The default setting of the search mode is set in the Environment. The FIND prompt line indicates only the non-default choice: L(IT or T(OK . If you do not specify a search mode, the default search mode (the one which is NOT mentioned on the prompt line) is used. To use Token search when the default is Literal search (prompt line says T(OK), type T after the prompt line and before the target string. To use Literal search when the default is Token search (prompt line says L(IT), type L before typing the target string. Note: nothing appears to happen when you type L or T ; the letter just appears where you are about to type the target string. See S(et E(nvironment in Miscellaneous Commands for more detail about Literal and Token search modes. In Literal search mode, the Editor will look for ANY occurrence of a string of characters that exactly matches the <TARGET> string. In Token search mode, the Editor will look for ISOLATED occurrences of the <TARGET> string. The Editor considers a string isolated if it is surrounded by any combination of delimiters, where a delimiter is any character that is not a number or letter.

Target string and delimiters

To allow the target string to contain any characters (including RETURN characters), the target string must be typed using special rules. In particular, the target string must be set off by characters called "delimiters". Both delimiters of the target string must be the same character. One delimiter must precede the first character of the string, and the same delimiter must follow the last character of the string.

The Editor allows any character which is not a letter or a number to be a delimiter. This lets you choose the delimiter. The most common choice is the slash (/) because it is a lower-case character that is not often found in the text, and it is easy to type. If you forget to precede the target string with a correct delimiter character, you will be told:

```
ERROR: INVALID DELIMITER.  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

Just try again, this time beginning with a correct delimiter.

ESC option

At any point during your response to the FIND prompt, you can abandon this command and return to the Edit level by pressing the ESC key.

S(ame-string option

Typing S instead of the delimited target string tells the Editor to use the same string that was last specified for the target string (the target string may have been specified either in Find mode or in Replace mode). From the Editor, typing the command

```
FS
```

will cause the cursor to jump to next occurrence of the previously specified target string. When the set direction specifies searching through the text in the reverse direction (toward the beginning of the file), FS may appear to have no effect. This is because Find mode places the cursor just AFTER the found occurrence of the target string. Unless the cursor is moved beyond the FIRST character of the previously found occurrence of the target string, FS will just keep finding the same occurrence of the target string again and again.

Note: The Environment (see S(et E(nvironment , in Miscellaneous Commands) displays the current <TARGET> string which will be invoked by typing S as a string response.

EXAMPLE 1:

Suppose you are editing a file containing the following text:

```
PROGRAM STRING1;
BEGIN
  WRITE('TOO WISE ');
  WRITE('YOU ARE');
  WRITELN(',');
  WRITE('TOO WISE ');
  WRITELN('YOU BE.')
END.
```

In the STRING1 program, with the cursor at the first P in the line

```
PROGRAM STRING1;
```

type F to select Find mode. When the FIND prompt line appears, type

```
'WRITE'
```

The two single quote marks (or two of some other delimiter) MUST be typed. The prompt line should now appear as:

```
>FIND[1]: L)IT <TARGET>  =>'WRITE'
```

When you type the last quote mark, the cursor jumps immediately to the first character following the E in the first occurrence of the Token target string

```
WRITE
```

EXAMPLE 2:

Again in the STRING1 program, with the cursor at the E of END. , type:

```
<2F
```

This prepares the system to Find the second pattern (you typed a repeat-factor of 2) in the reverse direction (you changed the set direction by typing <). When the prompt line appears, type

```
/WRITELN/
```

The prompt line should read:

```
<FIND[2]: L)IT <TARGET>  =>/WRITELN/
```

When you type the last / , the cursor will move immediately to the first character following the N in the second occurrence (searching backward through the file) of the Token string WRITELN .

EXAMPLE 3:

First, type

```
F/WRITE/
```

This locates the first occurrence of the Token string WRITE , searching in the set direction. Now, typing

```
FS
```

will make the prompt line flash:

```
>FIND[1]: L)IT <TARGET>  =>S
```

and the cursor will appear at the next occurrence of WRITE .

TEXT CHANGING COMMANDS

I(nsert

Insert mode is reached by typing I for I(nsert while at the Edit level. On entering Insert mode the following prompt line appears:

```
>INSERT: TEXT [<BS> A CHAR,<DEL> A LINE]  [<ETX> ACCEPTS, <ESC> ESCAPES]
```

Insert mode allows you to put new information into the text you are creating or editing. The characters that you type in this mode are inserted between the character on which you placed the cursor and the character that was immediately to the cursor′s left.

In order to maximize speed, the Editor does not constantly re-write the entire screen as you insert each new character. Instead, it makes a gap in the text, just where your insertion will appear, and then waits for you to type. Often you will have to terminate your insertion (by pressing CTRL-C) in order to see exactly how the insertion will look in its final form.

If you make a mistake while typing in Insert mode, just use the left-arrow key to backspace over your inserted characters. To delete the entire line that you are in the process of inserting, back to and including the previous RETURN character, type CTRL-X (some external terminals use the RUB or RUBOUT key, which generates ASCII DEL). The Insert prompt line helps you remember these mistake-correcting possibilities by "<BS> A CHAR" and "<DEL> A LINE". <BS> stands for the left-arrow (BackSpace) key and <DEL> stands for CTRL-X . You can erase only the text that you have inserted since entering Insert mode.

The set direction does not affect the Insert mode.

At any time during an insertion, you can cause the Editor to accept the insertion as it stands (making it a part of your file) by pressing CTRL-C (which the prompt line calls <ETX>). Until you press that CTRL-C you can cause the Editor to forget everything you have typed since entering Insert mode, by pressing the ESC key.

If an insertion is made and accepted (using CTRL-C), that insertion is also available (until the next insertion or deletion) in the Copy buffer, for use by the C(opy command. You can use this to duplicate your last insertion as many times as you wish. However, if <ESC> is used to reject the insertion, the Copy buffer is left empty.

The maximum size of a file is about 18400 bytes, or 38 diskette blocks. When your file can hold only another few hundred bytes, you may receive this warning as you begin typing in Insert mode:

```
ERROR: PLEASE FINISH UP THE INSERTION  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

When you respond by pressing the spacebar, you will still be in Insert mode. You can continue your insertion, but you have been warned that your file is almost full. You should start a new file right away or split the present file into two parts. If you continue typing in Insert mode, you will soon receive this more urgent message, when your file has exceeded the amount of text it can hold:

```
          ERROR: BUFFER OVERFLOW!!!!  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

Pressing the spacebar terminates your insertion, and any further attempts to initiate Insert mode cause this message:

```
          ERROR: NO ROOM TO INSERT.  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

Insert mode is immediately terminated, and you are not allowed to add any more text to your file.

Text Formats

There are two basic ways that text can be formatted as you Insert it. The formatting scheme is determined by the settings of various options in the Environment (see S(et E(nvironment, in Miscellaneous Commands). A(uto-indent is usually used for writing Pascal programs, while F(illing is most often used in writing text such as letters and other documents.

Inserting with A(uto-indent TRUE , F(illing FALSE

This is the normal setting of the Environment when you are writing Pascal programs. During an Insertion, the margins set in the Environment are ignored. Instead, you must terminate each line yourself, and start a new line, by pressing the RETURN key. Each new line automatically starts at the same indention as the first non-space character of the preceding text line.

A new indentation can be started by typing a space (to indent more) or by pressing the left-arrow key (to indent less) or by typing CTRL-Q (zero indentation) as the first character of any new line. The A(djust command can also be used to create a new indentation for a line.

If you use CTRL-C to terminate an insertion immediately after pressing the RETURN key (to start a new line), the cursor will automatically be indented the same amount as the line in which you began your insertion. This feature, which can be very useful in writing Pascal programs, ignores any change you may have made to the indentation of the insertion's first line, and ignores the indentation of intervening lines.

A paragraph cannot be formatted with the M(argin command while Auto-indent is set to True.

EXAMPLE:

With the Environment's A(uto-indent option set to TRUE, and the F(illing option set to FALSE, enter I(nsert mode and type the following sequence of keys (the names of special keys are enclosed in angle brackets <like this>):

```
          ONE<return>
          <space><space><space>TWO<return>
          THREE<return>
          <left-arrow>FOUR
```

This should create the indentations shown at the left below:

```
ONE            Original indentation
   TWO         Indentation changed by <space> <space> <space>
   THREE       <return> causes auto-indentation to level of line above
  FOUR         <left-arrow> changes indentation from level of line above
```

Inserting with A(uto-indent FALSE , F(illing TRUE

This is the normal setting of the Environment when you are writing text such as letters and other documents. It is the only Environment in which the M(argin command will function. The Editor forces all Insertions to be between the margins set in the Environment. The instant a new word (as you are typing it) exceeds the set R(ight margin, a RETURN character is automatically inserted before the word and the entire word (or as much of it as you have typed at that point) is placed beginning at the set L(eft margin. In the Editor, a "word" is any text character or characters bounded by any two "word delimiters", where a word delimiter is a space, a RETURN character, the beginning or end of the file, or the beginning or end of the current insertion (before CTRL-C is pressed). The hyphen is not a recognized word delimiter. If two or more RETURN characters are typed in succession, the next text is started at the set P(aragraph margin.

This setting of the Environment also causes the Editor to adjust the margins on the portion of the paragraph following an insertion (but not the paragraph portion preceding the insertion). The Editor considers a paragraph to be any text bounded by any two "paragraph delimiters", where a paragraph delimiter is a blank line (created by two RETURN characters), a line beginning with the C(ommand character (set in the Environment), or the beginning or end of the file.

Note: the automatic re-margining following an insertion can sometimes cause you much grief. If you are editing in or near a diagram, table, or other carefully formatted portion of text, it is a good precaution to temporarily set F(illing to False (just type SEFF<space>). This will prevent an incidental insertion from reformatting your beautiful diagram into a paragraph of meaningless text.

EXAMPLE:

With the Environment's A(uto-indent option set to FALSE, the F(illing option set to TRUE, the L(eft margin to 0, and R(ight margin to 10, enter I(nsert mode and type the following:

```
WISH I WEREN'T A WASH-AND-WEAR WARRIOR
```

This should create the text format shown at the left, below:

WISH I	Auto-returned when next word would exceed margin
WEREN'T A	Auto-returned when next word would exceed margin
WASH-AND-WEAR	Auto-returned at first possible break, even though beyond margin.
WARRIOR	

Inserting with A(uto-indent and F(illing both TRUE

With this setting of the Environment, A(uto-indent controls the left margin, ignoring the settings of the L(eft margin and P(aragraph margin. F(illing inserts RETURN characters as before, to keep lines from exceeding the set R(ight margin.

However, F(illing only operates to keep the CURRENT insertion from exceeding the Right margin. Any text on the same line, but to the right of the cursor, may extend beyond the Right margin or even beyond the 80 characters visible on the Apple Pascal system's display. The existence of characters beyond position 79 is indicated by an exclamation mark (!) displayed at the rightmost position on the screen. To see the hidden characters, insert a RETURN character anywhere in the visible portion of the line, or set A(uto-indent to False and M(argin the paragraph.

Changing the indentation can be done as before, by typing space, left-arrow, or CTRL-Q, but only if that is the FIRST character in a new line (not likely, since F(illing will generally begin a line with the last typed word). This setting of the Environment is not usually very useful, as its effects can be better obtained in other ways.

Inserting with A(uto-indent and F(illing both FALSE

With this setting of the Environment, the Editor ignores the margins set in the Environment. All margins, indentations and RETURN characters must be typed into the text by you. Characters may be inserted at any position on the screen.

If you attempt to type beyond position 71, the computer may beep to warn you. If you attempt to type beyond position 79, an exclamation mark (!) is displayed at the rightmost position on the screen. This character at the end of any line indicates that the line contains more than the 80 characters which can be displayed on the screen. Additional characters typed into that line are not lost, but they are not displayed. To see the hidden characters, you can Insert a RETURN character anywhere in the visible portion of the line; or you can set the Environment's F(illing option to TRUE, the A(uto-indent option to FALSE, and then issue the M(argin command.

D(elete

Delete mode is reached by typing D for D(elete while at the Edit level. On entering Delete mode, the following prompt line appears:

```
      >DELETE: < > <MOVING COMMANDS> [<ETX> TO DELETE, <ESC> TO ABORT]
```

In order to delete, the cursor must be in the correct position to begin the deletion. If you are going to delete to the right (forward through the text), place the cursor directly on the first character to be deleted. If you are going to delete to the left (backward through the text), place the cursor on the first character-position to the right of the first character to be deleted.

On typing D and entering Delete mode, the Editor remembers where the cursor is. That position is called the "anchor". As the cursor is moved away from the anchor in any direction, using the normal cursor-moves, all text between the cursor and the anchor disappears. When the cursor is moved toward the anchor, the "erased" characters re-appear. The repeat-factor may also be used to delete or undelete several lines at once, by prefacing a <RET> or any other cursor move with a repeat-factor while in Delete mode. The slash (/) repeat-factor cannot be used.

To accept the deletion at any point, use <ETX>. To undo the entire deletion at any time before using <ETX>, use <ESC>. Remember that <ETX> means to type CTRL-C and <ESC> means to press the ESC key.

Unlike inserting text, deleting text does NOT cause re-margining of the portion of the paragraph following the deletion, even if the Environment's F(illing option is set to TRUE and A(uto-indent is set to FALSE. Especially after a deletion that included a RETURN character, the line containing the cursor may extend beyond the 80-character limit of the Apple Pascal system's display. The invisible portion of the line is indicated by an exclamation mark (!) in the last visible character-position of the line. To see the rest of the line, insert a RETURN character anywhere in the visible portion of the line, or use the M(argin command to reformat the entire paragraph.

All the text between the cursor and the anchor position is stored in the Copy buffer, ready for use by the C(opy command, not only after you accept the deletion with CTRL-C , but also after you reject the deletion by pressing the ESC key. This last fact is useful when you want to duplicate some text in another location, or when you are moving some text to another location but wish to keep a backup copy of the text until the move is successfully completed.

If you attempt to delete too much text at one time (the maximum amount varies somewhat, depending on how large your file is at the moment.), the Copy buffer may be unable to hold all the deleted text. In that case, when you press CTRL-C to accept the deletion this message appears:

```
THERE IS NO ROOM TO COPY THE DELETION. DO YOU WISH TO DELETE ANYWAY? (Y/N)
```

If you type Y for "Yes", the text between the cursor and the anchor position is deleted but that text is not placed in the Copy buffer. If you type N for "No", the deletion is not carried out, and the text is not placed in the Copy buffer. After a response of either Y or N , the Copy buffer is left containing the same text it held before the D(elete command was initiated. If you reject a deletion that is too large for the Copy buffer, by pressing the ESC key, no message is given at that time. However, if you then attempt to C(opy from the Copy buffer this message appears:

```
          ERROR: NO ROOM  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

EXAMPLE:

Suppose you are editing the following text:

```
PROGRAM STRING2;
BEGIN
  WRITE('TOO WISE ');
  WRITELN('TO BE.')
END.
```

1) Move the cursor onto the E in END.
2) Type < (This changes the set direction to backward)
3) Type D to enter Delete mode.
4) Press the RETURN key twice. After the first RETURN the line WRITELN('TO BE.') disappears. After the second RETURN, the line WRITE('TOO WISE'); disappears.
5) Now press CTRL-C . The program after deletion appears as shown:

```
PROGRAM STRING2;
BEGIN
END.
```

The two deleted lines have been stored in the Copy buffer and the cursor has returned to the anchor position. Now use the C(opy command to copy the two deleted lines at any place to which the cursor is moved.

Note: after pressing CTRL-C , if you immediately C(opy the deletion withOUT moving the cursor, the deleted material is just replaced. This gives you one more chance to recover from a mistaken deletion.

Z(ap

The Zap command is executed by typing Z for Z(ap while at the Edit level. This command deletes all text between the current cursor position and the start of what was previously found, replaced or inserted.

The text position of the first character of the previous Find, Replace, or Insert is called the "equals mark". Typing the = key will place the cursor exactly at the equals mark, showing you where a Zap would end. You can then move the cursor (but do not use F(ind!) to the beginning of the material you wish to Zap.

This command is designed to be used immediately after one of the Find, Replace or Insert commands. If you Insert new material to the right of the old text that you want deleted, and then move the cursor back to the beginning of the old text and type Z, you will leave the Inserted material while deleting the old text.

If more than 80 characters are being Zapped, the Editor will ask for verification:

```
WARNING! YOU ARE ABOUT TO ZAP MORE THAN 80 CHARS, DO YOU WISH TO ZAP? (Y/N)
```

Repeat-factors and Zap: If a Find or a Replace is made with a repeat-factor, only the last string found or replaced will be deleted by Zap. All the other strings will be left as found or replaced.

All the text that is deleted by using the Zap command is placed in the Copy buffer, where it is available for use with the Copy mode (until the next insertion or deletion).

If you attempt to use Zap to delete too much text at one time (the maximum amount varies somewhat, depending on how large your file is at the moment), the Copy buffer may be unable to hold all the deleted text. In that case, when you type Z to Zap the deletion, first this message appears:

```
WARNING! YOU ARE ABOUT TO ZAP MORE THAN 80 CHARS, DO YOU WISH TO ZAP? (Y/N)
```

and then, when you type Y for "Yes", this message appears:

```
THERE IS NO ROOM TO COPY THE DELETION. DO YOU WISH TO DELETE ANYWAY? (Y/N)
```

If you type Y for "Yes", the text between the cursor and the "equals" position is deleted but that text is not placed in the Copy buffer. If you type N for "No", the deletion is not carried out, and the text is not placed in the Copy buffer. After a response of either Y or N , the Copy buffer is left containing the same text it held before the D(elete command was initiated.

C(opy

You get into Copy mode by typing C for C(opy at any time the EDIT prompt line is showing. On entering Copy mode, the following prompt line is displayed:

```
>COPY: B(UFFER F(ROM FILE <ESC>
```

Copying F(rom a Diskette File

To Copy text that is stored in another diskette file, so that it is inserted at the current cursor position in the file you are Editing (that is, between the character on which the cursor sits and the first character to the cursor's left), type C for C(opy and then type F for F(rom file and another prompt line appears:

```
>COPY: FROM WHAT FILE[MARKER,MARKER]?
```

Any existing diskette file may now be specified. You may type the filename's .TEXT suffix or not, as you wish. The suffix .TEXT is automatically supplied if you do not type it into your file specification. To suppress this feature (when C(opying from a file whose name does not end in .TEXT), type a period following the complete file specification. In order to Copy a portion of a file, two markers must have been set in the text of that file to bracket the desired text. The markers must have been set in the file at an earlier time, when that file was the file being Edited (see S(et M(arker , under Miscellaneous Commands).

If your response to the prompt line above does not include any marker names (in square brackets), the entire specified file is inserted into your workfile. If the file specification is followed by two marker names, enclosed in square brackets and separated by a comma, the portion of the specified file's text that is bounded by the two markers is inserted into your workfile. If [,marker] is used, the file is copied from the beginning to the marker. If [marker,] is used, the file is copied from the marker to the end of the file. Use of the Copy command does not change the contents of the file being copied from.

If your response to the prompt line above does not include any marker names (in square brackets), the entire specified file is inserted into your workfile. If the file specification is followed by two marker names, enclosed in square brackets and separated by a comma, the portion of the specified file's text that is bounded by the two markers is inserted into your workfile. If [,marker] is used, the file is copied from the beginning to the marker. If [marker,] is used, the file is copied from the marker to the end of the file. Use of the Copy command does not change the contents of the file copied from.

On the completion of the Copy command, after text has been copied from the specified diskette file, the cursor is placed on the first character of the text which was copied and this message appears:

```
BE SURE ORIGINAL SYSTEM.EDITOR DISK IS IN SAME DRIVE: {RETURN TO CONTINUE}
```

This message is for one-drive users who have copied from a file on a diskette other than their boot diskette. When the boot diskette is back in the boot drive, press the RETURN to continue.

If your present file can not contain all the additional text that you are attempting to Copy into it (maximum size of a file is about 18400 bytes, or 38 diskette blocks), the Editor copies in as much of the additional text as it can. Then it gives this message:

```
ERROR: BUFFER OVERFLOW.  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

When you press the spacebar, the Copy is complete; your file now contains as much of the additional text as the Editor could fit into your file.

EXAMPLE:

Suppose the diskette named MYDISK: contains a file named OLDFILE.TEXT, which has two markers placed in its text: ALPHA and BETA . Further suppose that you are now in the Editor, editing a new file, and you wish to insert at the current cursor position the text of OLDFILE.TEXT bounded by markers ALPHA and BETA .

In response to the EDITOR prompt line, you would first type a C to enter Copy mode, and then an F to select copying From-a-file. This prompt line would then appear:

```
>COPY: FROM WHAT FILE[MARKER,MARKER]?
```

To cause the planned insertion, type

```
MYDISK:OLDFILE[ALPHA,BETA]
```

Copying from the Copy B(uffer

Each time text is inserted or deleted, that text is also stored in the "Copy buffer", sometimes called the "insert-delete buffer". To use the text in the Copy buffer, type C to enter C(opy mode and then type B for B(uffer. The Editor immediately copies the contents of the Copy buffer into the file at the current location of the cursor (that is, between the character on which the cursor sits and the first character to the cursor's left). Use of the C(opy command does not change the contents of the Copy buffer.

On the completion of the C(opy command, after text has been copied from the Copy buffer, the cursor is placed on the first character of the text which was copied.

Unlike inserting text, Copying text does NOT cause re-margining of the portion of the paragraph following the Copy, even if the Environment's F(illing option is set to TRUE and A(uto-indent is set to FALSE. After Copying, some lines may extend beyond the 80-character limit of the Apple Pascal system's display. The invisible portion of the line is indicated by an exclamation mark (!) in the last visible character-position of the line. To see the rest of the line, insert a RETURN character anywhere in the visible portion of the line, or use the M(argin command to reformat the entire paragraph.

The Copy command can be used after an Insertion has been made, to duplicate the section of text just inserted, as many times as desired. Even more common is to use the Copy command to move text from one location in the file to another. Just D(elete the text from its present location, then move the cursor and C(opy the deleted text into its new location.

The contents of the Copy buffer are affected by the following commands:

1) D(elete: When you accept a deletion (with CTRL-C), the Copy buffer is loaded with the deleted text. When you reject a deletion (by pressing the ESC key), the Copy buffer is loaded anyway, with the text that would have been deleted had you accepted the deletion.

2) I(nsert: When you accept an insertion (with CTRL-C), the Copy buffer is loaded with the inserted text. When you reject an insertion (by pressing the ESC key), the Copy buffer is empty.

3) Z(ap: When you delete text using the Zap command, the Copy buffer is loaded with the deleted text.

The Copy buffer is of limited size (the actual size depends somewhat on how much of your computer's memory is occupied by your workfile). Whenever the proposed deletion (using either the Z(ap or the D(elete commnd) is greater than the amount of space available in the Copy buffer, the Editor will issue this warning:

```
THERE IS NO ROOM TO COPY THE DELETION. DO YOU WISH TO DELETE ANYWAY? (Y/N)
```

If you respond by typing Y for "Yes", the deletion is carried out in the normal way, but the deleted text is not stored in the Copy buffer. If you respond by typing N for "No", the deletion is rejected, but the rejected deletion is not stored in the Copy buffer. After either response, the contents of the Copy buffer remain what they were before D(elete or Z(ap was initiated.

If you Delete too much text and then reject the deletion by pressing the ESC key, you are given no message at that time. However, a subsequent attempt to Copy from the Copy buffer causes this rather enigmatic message:

```
ERROR: NO ROOM  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

and the Copy is not carried out.

If your file is already almost full (maximum file size is about 19900 bytes, or 42 diskette blocks), you may receive one of these messages when you attempt to C(opy from the Copy buffer:

```
ERROR: INVALID COPY.  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

or

```
ERROR: NO ROOM  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

In either case, the Copy is not carried out. This condition indicates that it is time to start a new file or to split your current file into two parts.

X(change

The eXchange mode is reached by typing X while at the Edit level. On entering eXchange mode the following prompt line appears:

```
>EXCHANGE: TEXT [<BS> A CHAR] [<ESC> ESCAPES; <ETX> ACCEPTS]
```

The eXchange mode is used to replace the character on which the cursor is sitting. As you type in eXchange mode, the cursor moves to the right along the line of text, replacing one character in the line each time you press a key. The left-arrow key (<BS>) can be used to move the cursor back one character, causing the character originally in that position (before the eXchange) to reappear. The set direction does not affect eXchange mode.

As with many other commands in the Editor, a text eXchange is made final by pressing CTRL-C (<ETX>). Pressing the ESC key (<ESC>) leaves eXchange mode without making any of the changes indicated since entering the mode.

Note: eXchange mode does not allow you to type beyond the end of the line, nor does it allow you replace a text character with a RETURN character.

EXAMPLE:

Suppose you wish to alter this line of text:

```
WRITE('TOO WISE ');
```

After placing the cursor on the W in WISE, type an X to enter eXchange mode. Now type the letter S and notice how it replaces the letter W . Press the left-arrow key to see the W reappear. Now type S again, and then M , leaving the line of text as follows:

```
          WRITE('TOO SMSE ');
```

Typing CTRL-C will make this change final, or pressing the ESC key will cause the original line to be retained.

R(eplace

Replace mode is reached by typing R for R(eplace while at the Edit level. On entering Replace mode, one of the following two prompt lines appears, depending on the setting of T(OKEN DEFault in the Environment (see S(et E(nvironment in Miscellaneous Commands):

```
          >REPLACE[1]: L(IT V(FY <TARG> <SUB>  =>
```

if the Environment's T(OKEN DEFault is set to TRUE, or

```
          >REPLACE[1]: T(OK V(FY <TARG> <SUB>  =>
```

if the Environment's T(OKEN DEFault is set to FALSE.

The Replace command searches through a file in the set direction to find repeat-factor occurrences of the specified TARGet string of characters, and replaces each of those occurrences (after verification, if that option is chosen) with the specified SUBstitute string of characters. When finished, it places the cursor at the end of the last string found and/or substituted.

Set direction

The R(eplace command searches for repeat-factor occurrences of the target string beginning at the present cursor position and scanning through the text in the set direction (indicated by the arrow at the beginning of the prompt line). An occurrence of the target string will be found only if it appears in that portion of the text which lies between the cursor and the end of the file toward which the search is progressing. See the section on the set direction (in this chapter, under General Information) in order to change the set direction arrow. If the end of the file is reached before the repeat-factor-th replacement can be carried out, this message appears:

```
    ERROR: PATTERN NOT IN THE FILE  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

Remember, however, that the search does not "wrap around". That portion of the file between the cursor and the end of the file in the direction OPPOSITE the set direction is not searched.

Repeat-factor

The repeat-factor is an integer from 0 to 9999 which may be typed just before typing the R for R(eplace. It is shown on the prompt line in square brackets: [1] , for example. If a repeat- factor of n is specified, the next n occurrences of the target string in the set direction are replaced. If no repeat-factor is specified, a repeat-factor of one is used. If a repeat-factor of / is used, all occurrences of the target string in the set direction are replaced.

L(iteral or T(oken search

The target string is treated somewhat differently, depending on whether Literal search or Token search is selected. The default setting of the search mode is set in the Environment. The REPLACE prompt line indicates only the non-default choice: L(IT or T(OK . If you do not specify a search mode, the default search mode (the one which is NOT mentioned on the prompt line) is used. To use Token search when the default is Literal search (prompt line says T(OK), type T after the prompt line and before the target string. To use Literal search when the default is Token search (prompt line says L(IT), type L before typing the target string. Note: nothing appears to happen when you type L or T ; the letter just appears where you are about to type the target string. See S(et E(nvironment in Miscellaneous Commands for more details about Literal and Token search modes.

V(erify option

The Verify option (shown as V(FY on the REPLACE prompt line) permits examination of each target string as it is found, before the replacement is carried out. You can then decide whether this occurrence of the target string is to be replaced or not. To select the Verify option in Replace mode, type V before typing the target string. Nothing will appear to happen when you type V, but the Verify option will be selected anyway. The following prompt line appears whenever Replace mode has found an occurrence of the target string in the file and Verify has been requested:

```
>REPLACE: <ESC> ABORTS, 'R' REPLACES, ' ' DOESN'T
```

Typing an R at this point will cause the specified replacement to be carried out, while pressing the spacebar will cause the Replace mode to search for the next occurrence of the target string, provided the specified repeat-factor (or the end of the file) has not been reached. The repeat-factor specifies the number of times an occurrence of the target string is to be found, not the number of times you actually type R to cause its replacement. Use / as the repeat-factor in order to examine every occurrence of the target string in the set direction.

Strings

The Editor has two string storage variables. The first string variable, called <TARGET> or <TARG> by the prompt line, contains the "target" string, and is used both by the F(ind command and by the R(eplace command. The target string is the sequence of characters which will be searched for by the Find command, or searched for and replaced by the Replace command. The second string, used only by the Replace command, is called <SUB> by the REPLACE prompt line and is the "substitute" string. In the Replace command only, the substitute string is the sequence of characters which will replace the target string when the target string is found.

String delimiters

To allow the target and substitute strings to contain any characters (including RETURN characters), each string must be typed using special rules. In particular, each string must be set off by characters called "delimiters". Both delimiters of a string must be the same character. One delimiter must precede the first character of the string, and the same delimiter must follow the last character of the string.

The Editor allows almost any normal printing character which is not a letter or a number to be a delimiter. This lets you choose the delimiter. The most common choice is the slash (/) because it is a lower-case character that is not commonly found in the text, and it easy to type.

Once you have typed the initial delimiter character for either the target or the substitute string, you cannot backspace (using the left-arrow key) to erase that character or any of the preceding characters in your response. If you forget to precede either the target string or the substitute string by a correct delimiter character, you will be told.

```
ERROR: INVALID DELIMITER.  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

You will get the same message if you try to backspace (by pressing the left-arrow key) immediately after typing the target string's final delimiter. Just try the whole command again, and this time use the correct delimiters.

Note: many CTRL characters have other system uses, and should not be used as string delimiters. These include CTRL-A (screen page-flip), CTRL-F (stop output), CTRL-H (left-arrow key), CTRL-I (tab), CTRL-M (RETURN), CTRL-S (stop program), CTRL-Z (cursor auto-follow), and CTRL-@ (hangs system).

ESC option

At any time during your response to the REPLACE prompt, you can abandon this command and return to the Edit level by pressing the ESC key.

The Same-string option

Typing S in the place of the delimited target string tells the Replace command to use the same target string that was last specified. The target string may have been specified either by the Find command or by a previous use of the Replace command. Similarly, typing S in the place of the delimited substitute string tells the Replace command to use the same substitute string that was last specified by a previous use of the Replace command. For example, in Replace mode, typing

```
S/<any-string>/
```

causes the Replace mode to use the previous target string (and a new substitute string), while typing

```
/<any-string>/S
```

causes the previous substitute string to be used (and a new target string). From the Editor, typing the command

```
RVSS
```

says "Do it again": it causes the next occurrence of the previously specified target string to be replaced (after verification) with the previously specified substitute string.

Note: when the set direction specifies searching through the text in the reverse direction (toward the beginning of the file), RVSS may appear to have no effect if you chose NOT to replace the last found occurrence of the target string. This is because the Replace command places the cursor just AFTER the found occurrence of the target string. Unless the cursor is moved beyond the first character of the currently found occurrence of the target string, or unless that occurrence is changed to a different string, RVSS will just find the same occurrence of the target string again and again.

Note: The Environment (see S(et E(nvironment, in Miscellaneous Commands) shows you the current <TARGET> and <SUBST> strings which will be invoked by typing S as a string response.

EXAMPLE 1:

Suppose you wish to replace the next three occurrences of the target string APPLE with the substitute string BANANA :

From Edit mode, you would type

```
3R
```

to indicate a repeat-factor of 3 and then to select the Replace mode. In response to the REPLACE prompt line:

```
>REPLACE[3]: T(OK V(FY <TARG> <SUB>  =>
```

you could type

```
/APPLE/)BANANA)
```

In this example, first the character / is used as the beginning and ending delimiter for the target string, and then the character) is used as the beginning and ending delimiter for the substitute string. In the example, two different delimiters were used for pedagogical purposes. In practice you would be more likely to use

```
/APPLE//BANANA/
```

If you now wish to Replace 5 more occurrences of the target string APPLE , but this time with the substitute string PAPAYA , just type (from Edit mode)

```
5RS/PAPAYA/
```

After a brief flash of this prompt line

```
>REPLACE[5]: T(OK V(FY <TARG> <SUB>  =>S/PAPAYA/
```

the requested replacements will be carried out.

EXAMPLE 2:

From Edit mode, if you type

```
RL/QX//YZ/
```

this prompt line should appear:

```
>REPLACE[1]: L)IT V)FY <TARG> <SUB>  =>L/QX//YZ/
```

This command will change the program line

```
          VAR SIZEQX:INTEGER;
```

to

```
          VAR SIZEYZ:INTEGER;
```

You must select the non-default Literal search mode (by typing L before typing the target string) because the string QX is not a token but is part of the token SIZEQX.

FORMATTING COMMANDS

A(djust

Adjust mode is reached by typing A for A(djust while at the Edit level. On entering Adjust mode, the following prompt line appears:

```
>ADJUST: L(JUST R(JUST C(ENTER <LEFT,RIGHT,UP,DOWN-ARROWS> [<ETX> TO LEAVE]
```

The Adjust mode is designed to make it easy to adjust the indentation of a line or a whole group of lines. Cursor moves (using the right-arrow and the left-arrow keys) can be used to push the line right and left, or you can adjust the line to the L(eft margin, the R(ight margin, or the C(enter. Moving the cursor up or down makes the same adjustment to lines above or below. Use of a repeat-factor is valid with all cursor moves.

Once you are in Adjust mode, each time the right-arrow key is typed, the whole line moves one space to the right. The line can be moved beyond the Right margin set in the Environment. Characters moved beyond the 80-th character position are not displayed, but their existence is indicated by an exclamation mark (!) in the 80-th character position of the line.

Each time the left-arrow key is typed, the whole line moves one position to the left. The line can be moved beyond the Left margin set in the Environment, but the leftmost character cannot be moved beyond the left edge of the screen display (character position zero).

When the line is adjusted to the desired indentation press <ETX> (that's CTRL-C , of course).

Note: <ESC> cannot be used to cancel an Adjustment. You MUST accept the adjustment, by pressing CTRL-C .

In order to adjust a whole sequence of lines, first adjust the top or the bottom line, then (BEFORE typing CTRL-C) use CTRL-O or CTRL-L commands and the line above (or below) will automatically be adjusted by the same amount when the cursor jumps to that line. Finally, when the entire sequence has been adjusted, type CTRL-C.

Repeat-factors, including / , are valid when used before any of the cursor moves while in Adjust mode.

Adjust mode can also be used to center text on the page and to left-justify or right-justify text (force all the lines to make a smooth left margin, like this page, or a smooth right margin). Typing L for L(JUST while in Adjust mode causes the line containing the cursor to be left-justified by moving the leftmost non-space character to the Left margin set in the Environment. Similarly, typing R for R(JUST right-justifies the line by moving the rightmost text character to the set Right margin. Typing C for C(ENTER causes the line to be centered between the set Left and Right margins. Typing CTRL-O or CTRL-L (before CTRL-C is typed) will cause the line above (or below) to be adjusted to the same specification (left-justified, right-justified or centered) as the previously adjusted line.

M(argin

The Margin command is executed by typing M for M(argin while at the Edit level. There is no indication of this command in the EDIT prompt line. Within the paragraph containing the cursor, the Margin command adjusts the text to compress it as much as possible without violating the three margins set in the Environment.

Margin is an Environment dependent command; that is, it may only be executed when F(ILLING is set to TRUE and A(UTO INDENT is set to FALSE in the Environment. If you attempt to Margin a paragraph when F(illing and A(uto indent are not set correctly in the Environment, this message appears:

```
    ERROR: INAPPROPRIATE ENVIRONMENT  PLEASE PRESS <SPACEBAR> TO CONTINUE.
```

You must set these two options correctly in the Environment before the Margin command can be executed. There are also three parameters (all are Set in the Environment) used by this command: R(IGHT MARGIN, L(EFT MARGIN and P(ARAgraph MARGIN. See S(et E(nvironment under Miscellaneous Commands for how to set F(illing, A(uto-indent, and the margin values.

The Margin command affects only the paragraph which contains the cursor. A paragraph is defined to be any text bounded above and below by any two paragraph delimiters, where a paragraph delimiter may be a blank line (created by two consecutive RETURN characters), the beginning of the file, the end of the file, or a line which starts with the Command character that is currently set in the Environment. Unless you change it (see S(et E(nvironment), the COMMAND CHaracter is set by default to the carat (^).

To Margin a paragraph, move the cursor to anywhere in that paragraph and type M . When doing an exceptionally long paragraph, it may take several seconds before the routine is ready to redisplay the screen. When breaking lines to avoid exceeding the right margin, the Margin command recognizes all spaces as possible points to break the line. All other characters in sequence are considered words, and will not be broken. The Margin command does not recognize hyphens as possible line break points, nor does it know how to correctly introduce hyphens into words that do not already contain them.

Certain characters or character combinations, when followed by one or more spaces, will be followed by exactly two spaces after a Margin command. These characters include the following: period, question mark, colon, exclamation point, or any of those characters immediately followed by a close-parenthesis or double quote.

EXAMPLE:

The paragraph below has been Margined with these Environment parameters:

L(eft margin	0
R(ight margin	72
P(aragraph margin	8

```
------------------------------------------------------------------------
        When you operate a skateboard in excess of 350 miles per hour,
certain problems are encountered.  First of all, the number of traffic
citations becomes excessive, unless your skateboard is equipped with
the proper racing stripes.  Secondly, goggles and knee protectors
often blow away and skateboards have been known to become airborne.
Lastly, you may have to endure the ire of Porshe and Ferrari drivers,
since they become depressed, angered, and sometimes say uncomplimentary
things when passed by a person on a skateboard.

------------------------------------------------------------------------
```

Next, the same paragraph is shown after being Margined with these parameters set in the Environment:

```
L(eft margin        10
R(ight margin       64
P(aragraph margin    0
```

```
When you operate a skateboard in excess of 350 miles per hour,
          certain problems are encountered.  First of all, the
          number of traffic citations becomes excessive, unless
          your skateboard is equipped with the proper racing
          stripes.  Secondly, goggles and knee protectors often
          blow away and skateboards have been known to become
          airborne.  Lastly, you may have to endure the ire of
          Porshe and Ferrari drivers, since they become
          depressed, angered, and sometimes say uncomplimentary
          things when passed by a person on a skateboard.
```

MISCELLANEOUS COMMANDS

S(et

Set mode is reached by typing S for S(et while at the Edit level. There is no indication of the Set mode option on the EDIT prompt line. On entering the Set mode, the following prompt line appears:

```
>SET: E(NVIRONMENT M(ARKER <ESC>
```

S(et M(arker:

When you are editing a large file, it is particularly convenient to be able to jump directly to certain places in the file by using markers that have been set in the desired places. Once set, it is possible to jump to these markers at any time, by using the M(arker option in the J(ump mode (see Moving Commands).

The C(opy F(rom File command can also make use of markers that have been placed in the text of a file. When you are editing one file, the marked portion of a second file that is stored on diskette may be copied into the file you are editing (see Text Changing Commands).

This is how you place a marker in the text of a file that you are editing. While still in Edit mode, move the cursor to the spot in the text where you want the marker to be placed. When the cursor is in the desired spot, type S to enter S(et mode, and then type M for M(arker. The following prompt line appears:

```
SET WHAT MARKER?
```

This is asking you to type the name of the marker which will be placed at the current cursor position. The marker name may be up to eight characters (if you type more, they will be truncated to the first eight), terminated by pressing the RETURN key. Almost any character except a carriage return may be used in a marker name, but all lower-case letters are converted to upper-case letters. Note that the C(opy F(rom file command uses a comma to separate the two markers which specify the text to be copied. For use with that command, the first marker name must not contain a comma. If a marker with the specified name has already been placed in the text at an earlier time, the old marker is moved to the current cursor position without comment, and the old position is lost.

Only ten markers are allowed in a file at any one time. If you attempt to place an eleventh marker, the following message appears:

```
MARKER OVFLW.  WHICH ONE TO REPLACE?
0) name1
1) name2
.  ...
.  ...
9) name10
```

You must eliminate one of your existing markers before you can place the new one. Choose a number from 0 through 9, type that number and its place in the list will now be available for your new marker name. You can use this method to rename or re-place an existing marker, but you can never simply remove a marker from your file, even if you delete all the text that contained the marker.

A marker specifies an absolute position in the file. If an insertion, copy or deletion is made between the beginning of the file and the marker position, the marker will move along with the associated text although the movement is only approximate.

S(et E(nvironment

The Editor lets you set various aspects of the editing "environment" to suit the task at hand. From the Edit level, type S to enter the S(et mode, and then type E for E(nvironment. The screen display is replaced with a prompt similar to the one shown below:

```
>ENVIRONMENT: [OPTIONS] <ETX> OR <SP> TO LEAVE
    A(UTO INDENT  TRUE
    F(ILLING      FALSE
    L(EFT MARGIN  0
    R(IGHT MARGIN 79
    P(ARA MARGIN  5
    C(OMMAND CH   ^
    T(OKEN DEF    TRUE

    7436 BYTES USED,  12020 AVAILABLE.

    PATTERNS:
      <TARGET>= 'APPLE', <SUBST>=  'BANANA'

    MARKERS:
        START            PART3            SUMMARY
        INTRO            MAINPARA         BIBLIOG
        ACKNOWL          PART 5           INDEX

    DATE CREATED: 4-13-79   LAST USED:  7-28-79
```

By typing the appropriate first letter, any or all of the options listed in the upper portion of the display may be changed. The settings shown for many of the options are the default settings for the Editor on most screens. Implementations for some external terminals may use different defaults.

The portion of the display showing the PATTERNS: <TARGET> and <SUBST> will not appear unless you have used the F(ind or R(eplace commands since entering the Editor this time. The portion of the display showing the MARKERS: currently in the file will not appear unless you have at some time used the S(et M(arker command to place a marker in the text.

The information stored in the Environment (with the exception of the <TARGET> and <SUBST> strings) is saved each time you save the file on diskette, so the system can "remember" that environment each time you work on that file again.

THE ENVIRONMENT OPTIONS

A(uto indent

Auto-indent affects only the Editor commands I(nsert (under Text Changing Commands) and M(argin (under Formatting Commands). See the discussions of those commands for more details and examples.

The A(UTO INDENT option is set to TRUE (each new line is automatically started at the same indentation as the first non-space character of the previous line) by typing A T .

The A(UTO INDENT option is set to FALSE (new lines begin at the screen's left edge or at the set Left margin and Paragraph margin) by typing A F . Unless Auto-indent is False (and Filling is True), the Margin command will not operate and the Insert command will not cause re-marginning of the portion of a paragraph following an insertion.

Auto-indent should generally be True for writing and editing Pascal programs, and False for writing and editing natural language text.

F(illing

Filling affects the Editor's I(nsert Command (under Text Changing Commands) and allows the M(argin command (under Formatting Commands) to function. See the discussions of those commands for more details and examples.

The F(ILLING option is set to TRUE (lines are automatically broken between words -- at spaces and hyphens -- to avoid exceeding the set Right margin) by typing F T . Unless Filling is True (and Auto-indent is False), the Margin command will not operate and the Insert command will not cause re-marginning of the portion of a paragraph following an insertion.

The F(ILLING option is set to FALSE (the set margins are ignored; you must end each line yourself) by typing F F .

Filling should generally be False for writing or editing Pascal programs, and True for writing or editing natural language text. However, if you are editing a table, diagram, or other carefully formatted portion of text, it is a very good safety precaution to set Filling to False (from the Edit level, just type SEFF<space>). This will save you the frustration of having your text completely re-formatted following an insertion.

L(eft margin
R(ight margin
P(aragraph margin

When Filling is True (and Auto-indent is False), the margins set in the Environment are the margins which are used by the I(nsert command (under Text Changing Commands) and the M(argin command (under Formatting Commands). These margins also affect the Center, Left, and Right justifying commands in the Adjust mode (under Formatting Commands). See the discussions of those commands for more details and examples.

To change the value for the L(EFT MARGIN option, type L followed by an unsigned integer, and then press the spacebar. The unsigned integer may also be terminated by pressing the RETURN key. The value that you type replaces the old value for the Left margin in the prompt display shown at the beginning of this section.

To change the value for the R(IGHT MARGIN option, type R followed by an unsigned integer, and then press the spacebar. Similarly, you can change the value of the P(ARA MARGIN option by typing P followed by an unsigned integer, and then press the spacebar.

All unsigned integers with four or fewer digits are valid margin values. If you attempt to assign a margin value of more than four digits, the value will be truncated to the first four digits typed. To create normal text displays whose characters are all visible on the screen, you should use margin values from 0 through 79, and the Left and Paragraph margin values should be less than the value of the Right margin. To create text that is confined to the left "page" of the 40-character Apple screen, use margin values from 0 through 39.

C(ommand character

The Command character affects the M(argin command (under Formatting Commands) and re-marginning in the Insert mode (under Text Changing Commands). See the discussions of those commands for more details.

To change the setting of the C(OMMAND CH option, type C followed by almost any character except the RETURN character or ESCape. For example, typing C * will change the set Command character to * . This change will be reflected in the Environment prompt.

If the Command Character appears as the first non-blank character in a line of text, then that line is protected from the Margin command, and from re-marginning following an Insertion. That line is also treated as a paragraph delimiter for marginning purposes. The normal Command character is the carat or circumflex accent (^). Unless you have some special use for the carat character in your text, you should generally leave it as the set Command character.

T(oken default

This option affects the search mode used by the commands F(ind and R(eplace (under Text Changing Commands). See the discussions of those commands for more details and examples.

The T(OKEN DEF option is set to TRUE (the default search mode is Token search) by typing T T , and to FALSE (the default search mode is Literal search) by typing T F .

In Literal search mode, the Editor will look for ANY occurrence of a string of characters that exactly matches the <TARGET> string. In Token search mode, the Editor will look for ISOLATED occurrences of the <TARGET> string. The Editor considers a string isolated if it is surrounded by any combination of delimiters, where a delimiter is any character that is not a number or letter.

For example, in the sentence "Put the book in the bookcase.", using the <TARGET> string "book", Literal search mode will find two occurrences of "book" while Token search mode will find only one, the word "book" isolated by the delimiters <space> <space>.

In Token search mode you can find an occurrence of the <TARGET> string, even if the occurrence has more spaces or fewer spaces (including zero) corresponding to each space in the specified <TARGET> string. For example, suppose you are searching the following text, which contains four slightly different occurrences of the words "APPLE PIE":

```
I'LL HAVE SOME A PPLEPIE, SOME APPLE    PIE,
SOME APPLEPIE, AND THEN SOME AP PLE PIE, TOO.
```

If you use the <TARGET> string "APPLEPIE" , a Token search will find only the third occurrence. With the <TARGET> string "APPLE PIE" , a Token search will find both the second occurrence (which has more spaces, but at the right place in the string) and the third occurrence (which has fewer spaces, and none in the wrong place). With the <TARGET> string "A P P L E P I E" , a Token search will find all four occurrences.

However, only a Literal search would find an occurrence of "APPLE PIE" that was buried in the word "CRABAPPLE PIE". That's because the "B" would not constitute a proper isolating delimiter.

When editing natural language text, it is a good idea to use Literal search mode (set T(OKEN DEFault to FALSE). When editing programs, it is usually more useful to use Token search mode (leave T(OKEN DEF set to TRUE).

V(erify

The Verify command is executed by typing V for V(erify while at the Edit level. There is no indication of the V(erify command on the EDIT prompt line. The status of the Editor is verified by redisplaying the screen. The Editor attempts to adjust the window so that the cursor is at the center of the screen. This command can be psychologically helpful. Type it whenever you are unsure that the screen really corresponds to what is in your file, especially when an Insertion leaves an extra word or two dangling beyond the right end of a line. After typing V the screen is pretty much guaranteed to accurately reflect what is really in your file.

Q(uit

Quit mode is reached by typing Q for Q(uit while at the Edit level. On entering Quit mode, the screen display is replaced by this prompt message:

```
>QUIT:
    U(PDATE THE WORKFILE AND LEAVE
    E(XIT WITHOUT UPDATING
    R(ETURN TO THE EDITOR WITHOUT UPDATING
    W(RITE TO A FILE NAME AND RETURN
    S(AVE WITH SAME NAME AND RETURN
```

One of the four displayed options (described below in more detail) must be selected by typing U for U(pdate, E for E(xit, R for R(eturn, W for W(rite, or S for S(ave .

U(pdate

This tells the Editor to erase all previous versions of the boot diskette's workfile (SYSTEM.WRK.CODE as well as SYSTEM.WRK TEXT). Then it saves on the boot diskette, under the filename SYSTEM WRK TEXT, a backup copy of the file currently in memory. An U(pdate should be done at least every 15 minutes, in order to prevent accidental loss of your efforts. From the Editor, every so often, just type Q U E . In a few seconds, the boot diskette's file SYSTEM.WRK.TEXT will contain the latest version of your workfile, and you will again be in the Editor, ready to continue working on your backed-up workfile.

E(xit

This causes the system to leave the Editor without saving on diskette the file that is currently being worked on in memory. The boot diskette's backup copy of the workfile, SYSTEM.WRK.TEXT, is not updated to contain the latest version of the workfile. This means that any modifications made since entering the Editor are not recorded in SYSTEM.WRK.TEXT or in any other place. All editing which was done during the session is irretrievably lost. After selecting this option, you are placed at the Command level.

R(eturn

This option lets you return directly to the Editor without updating. The cursor is returned to the exact place in the file it occupied when Q was typed. Usually this command is used after unintentionally typing Q .

W(rite

Selecting this option causes a further prompt to be displayed:

```
>QUIT:
NAME OF OUTPUT FILE (<CR> TO RETURN) -->
```

The file in memory may now be saved under any diskette filename. You do not need to specify the .TEXT suffix; it will be supplied automatically. If you do not want .TEXT added to the filename, type a period following the file specification. In response to the above prompt, you can also type your filename, complete with .TEXT suffix, followed by file size specification. This lets you store your file in a space other than the largest unused area on the diskette.

If you wish to return directly to editing the file currently in memory, without saving it, just press the RETURN key instead of typing a filename. If a file with the specified filename already exists on the specified diskette, you will be warned. For example, if you try to write to MYDISK:MYFILE, and that file already exists on MYDISK: you will be asked:

```
REMOVE OLD MYDISK:MYFILE.TEXT?
```

Type "Y" to continue. After your file has been saved on diskette, the Editor displays a message similar to this:

```
>QUIT:
WRITING.....
YOUR FILE IS 1978 BYTES LONG.
DO YOU WANT TO E(XIT FROM OR R(ETURN TO THE EDITOR?
```

If you type E for E(xit, the system leaves the Editor and returns to the Command level. If you type R for R(eturn, you are again placed in the Editor, with the cursor in the same position it occupied in the file when Q was typed.

S(ave

When you choose this option, your new file will have the same name as the file that was most recently called by the Editor. If the file you S(ave was created at the current editing session and therefore does not yet have a name, the file will be S(aved as SYSTEM.WRK.TEXT .

After you invoke the S(ave option, you will be asked if you want to purge your original file. For example, if you G(et MYDISK:MYFILE from the Filer, edit the file, Q(uit the Editor and then S(ave the updated text, the following message will appear:

```
          PURGE OLD MYDISK:MYFILE BEFORE S(AVE?
```

If you type "Y" the old file will be removed from the disk before the new file is written out. This may cause the new file to overwrite the old file. If you have no backup of the original file and it is a large file, it would be safer to type "N". When you type "N" the old file will not be overwritten and only will be removed when the new file is successfully written to the disk. If there is not room to copy the new file before destroying the old one, the message

```
ERROR:  WRITING OUT THE FILE.  PLEASE PRESS <SPACEBAR> TO CONTINUE
```

will appear. Pressing the spacebar will return you to the Editor.

Do not press RESET after you have given the system permission to purge your original file; doing so may destroy both the old and new versions of your file.

After your file has been saved on diskette, the Editor displays a message similar to this:

```
          >QUIT:
          WRITING.....
          YOUR FILE IS 1978 BYTES LONG.
          DO YOU WANT TO E(XIT FROM OR R(ETURN TO THE EDITOR?
```

If you type E for E(xit, the system leaves the Editor and returns to the Command level. If you type R for R(eturn, you are again placed in the Editor, with the cursor in the same position it occupied in the file when Q was typed.

EDITOR COMMAND SUMMARY

SCREEN COMMANDS

CTRL-A	Shows the other 40-character "page" of the display.
CTRL-Z	Screen scrolls right and left to follow the cursor.

SPECIAL CHARACTERS

CTRL-K	Produces left bracket: [
SHIFT-M	Produces right bracket:]

CURSOR MOVES

right-arrow key	Moves repeat-factor spaces right.
left-arrow key	Moves repeat-factor spaces left.
CTRL-O	Moves repeat-factor lines up.
CTRL-L	Moves repeat-factor lines down.
spacebar	Moves repeat-factor spaces in set direction.
CTRL-I	Moves repeat-factor tab positions in set direction.
RETURN key	Moves to start of line that is repeat-factor lines away, in set direction.
=	Moves to start of latest text found, replaced or inserted.

REPEAT-FACTOR

An integer from 0 through 9999 typed before a move or command. If repeat-factor is / the move or command is repeated as many times as possible in the file.

SET DIRECTION

< , -	All change set direction to backward
> . +	All change set direction to forward

MOVING COMMANDS

J(ump: Jumps to file's B(eginning or E(nd, or to previously set Marker.

P(age: Moves the cursor repeat-factor pages in the set direction.

F(ind: Looks in the set direction for the repeat-factor-th L(iteral or T(oken occurrence of the <TARGET> string, which must be typed with delimiters. S means use the same string as before.

TEXT CHANGING COMMANDS

I(nsert: Inserts text. Use left-arrow key to backspace over insertion. CTRL-Q deletes back to the most recent RETURN character in the current insertion. CTRL-X acts like CTRL-Q except that it also deletes the RETURN.

D(elete: Deletes all text moved over by the cursor. Back up cursor to undelete.

Z(ap: Deletes all text between the current cursor position and the "equals position" (at the start of the latest text found, replaced or inserted).

C(opy: Copies a diskette file, or what was last inserted, deleted or zapped, into the file at the position of the cursor.

X(change: Replaces the character under the cursor with the character typed. Each line must be done separately. Pressing the left-arrow key causes the original character to re-appear.

R(eplace: Looks in the set direction for the next L(iteral or T(oken occurrence of <TARG> string, and replaces it with <SUB> string. Continues repeat-factor times. Both strings must be typed with delimiters. V(erify option asks for permission to replace. S means use the same <TARG> or the same <SUB> string as before.

FORMATTING COMMANDS

A(djust: Adjusts indentation of the line the cursor is on. Left-arrow and right-arrow keys move the line left and right. Moving cursor up or down adjusts lines above or below by same amount.

M(argin: Starting at the cursor position, adjusts all text between two blank lines (one paragraph) to the margins which have been set.

MISCELLANEOUS COMMANDS

S(et: Sets a M(arker of the specified name at the current cursor position. Sets options in the E(nvironment for A(uto-indent, F(illing, margins, default search mode, and C(ommand characters.

V(erify: Redisplays the screen with the cursor centered.

Q(uit: Leaves the Editor. You may U(pdate the workfile, E(xit without updating, R(eturn to the Editor, W(rite to any diskette file, or S(ave to your original file.

CHAPTER 5

THE PASCAL COMPILER

128 INTRODUCTION
128 Diskfiles Needed
130 USING THE COMPILER

INTRODUCTION

The basic purpose of the Apple Pascal Compiler is to convert the text of a Pascal program into the compressed P-code version of the program. This P-code is the "machine language" of the Pascal pseudo-machine, or "P-machine", described in this manual's appendices on the P-MACHINE. The P-code version of your program can then be run on virtually any computer for which the P-machine interpreter has been implemented.

The compiler control options recognized by the Apple Pascal Compiler are described in the Apple Pascal Language Reference Manual. That manual also contains a list of the error messages reported by the Apple Pascal Compiler.

This chapter is written specifically for using the Apple Pascal operating system with the Apple Pascal programming language and the Apple Pascal Compiler. If you are using any other programming language, you must first read that language's reference manual for special instructions on using this operating system with that language.

DISKFILES NEEDED

The following diskfiles allow you to invoke the Compiler explicitly:

Textfile to be Compiled	(any diskette, any drive; default is boot diskette's text workfile SYSTEM.WRK.TEXT, any drive; Pascal compiler $I option may also specify other source textfiles)
SYSTEM.COMPILER	(any diskette, any drive; required)
SYSTEM.LIBRARY	(boot diskette, boot drive; required only if program USES Intrinsic Units; Pascal compiler $U option may also specify other library files)
SYSTEM.EDITOR	(any diskette, any drive; optional; to fix errors found by Compiler)
SYSTEM.SYNTAX	(boot diskette, any drive; optional messages given on entering Editor)

In addition to the above files, the following files may be needed if you are invoking the Compiler automatically via the R(un command:

SYSTEM.LINKER	(any diskette, any drive; required if external routines need to be linked)
SYSTEM.LIBRARY	(boot diskette, any drive; required to contain needed external routines, if Linker called)

SYSTEM.PASCAL	(boot diskette, boot drive; required between Compiling, Linking and eXecuting steps)
SYSTEM.LIBRARY	(boot diskette, boot drive; required if program uses long integers, does file I/O using real numbers or SEEK, or USES Intrinsic Units)
SYSTEM.CHARSET	(any diskette, any drive; required only if program uses WCHAR or WSTRING from TURTLEGRAPHICS)

When you type C for C(ompile from the Command level, the file SYSTEM.COMPILER must be available on a diskette in any on-line disk drive. After the Compiler reports an error, if you elect to re-enter the Editor by typing E for E(dit from the Compiler, the file SYSTEM.EDITOR must be available in any disk drive.

On entering the Editor, a message describing your program's error will be given if the file SYSTEM.SYNTAX is available on the boot diskette. SYSTEM.SYNTAX is optional; if it is not on the boot diskette the error is reported by number. Compiler error messages corresponding to these numbers are given in an appendix at the end of the Apple Pascal Language Reference Manual. Some users will wish to save room on their boot diskettes by removing the file SYSTEM.SYNTAX, referring to the table of messages in the appendix, instead.

One-drive note: The files SYSTEM.COMPILER , SYSTEM.EDITOR , and SYSTEM.SYNTAX are all on diskette APPLE0: , which is the normal one-drive boot diskette. If you have been working on a program in the Editor, and U(pdating the workfile, your boot diskette has all the files needed to R(un or C(ompile the workfile. If you wish to R(un or C(ompile a textfile that is not already on the boot diskette, use the Filer's T(ransfer command to transfer that textfile onto your boot diskette before compiling. If your program requires Linking to external routines, see this manual's chapter THE LINKER for help.

Multi-drive note: The files SYSTEM.EDITOR and SYSTEM.SYNTAX are both on diskette APPLE1: , which is the normal multi-drive boot diskette. The file SYSTEM.COMPILER is on diskette APPLE2: , which is normally kept in drive volume #5: in a multi-drive system. With APPLE1: in the boot drive and APPLE2: in a non-boot drive, your system has all the files needed to R(un or C(ompile the workfile.

Two-drive note: If you wish to R(un or C(ompile a textfile that is not already on APPLE1: or APPLE2: , and your system has only two drives, use the Filer's T(ransfer command to transfer that textfile onto either APPLE1: or APPLE2: before compiling. This T(ransfer is not necessary on systems with three or more drives. Another possibility for two-drive systems is to make APPLE0: your boot diskette (just put APPLE0: in the boot drive and press the Apple's RESET key). This frees your second drive to hold a source or destination diskette for compilations, saving you from T(ransferring the source file onto APPLE1: or APPLE2: . APPLE0: does not contain SYSTEM.LINKER; if your program requires Linking to external routines, use APPLE1: and APPLE2: .

If a compilation is so large that the Apple's available memory is insufficient, you can use the Filer to M(ake a four-block file called SYSTEM.SWOPDISK on the diskette containing SYSTEM.COMPILER. Before any attempt to read a new diskette directory (when finding an Include-file for example) the system will first temporarily store the bottom 2K bytes of the data heap in SYSTEM.SWOPDISK and then put the diskette directory into that 2K area of the Heap. This saves 2K of top-of-Heap memory. When the directory is no longer needed, the contents of SYSTEM.SWOPDISK are stored back into their original Heap locations.

USING THE COMPILER

The Compiler is invoked by typing C for C(ompile or R for R(un from the outermost Command level of the Apple Pascal operating system. The screen immediately shows the message

```
COMPILING...
```

The Compiler automatically compiles the boot diskette's workfile SYSTEM.WRK.TEXT or another workfile designated by the Filer's G(et command and saves the resulting code (if compilation is successful) as SYSTEM.WRK.CODE . If there is a workfile, but you do not wish to compile that file, use the Filer's N(ew command to clear away the workfile before compiling. If no workfile is available, you are prompted for a source filename:

```
COMPILE WHAT TEXT?
```

You should respond by typing the name of the text file that you wish to have compiled.

It is not necessary to type the suffix .TEXT ; that suffix is automatically supplied by the Compiler. If you wish to defeat this suffix-adding feature, to compile a textfile whose filename does not end in .TEXT , type a period (.) after the last character of your filename.

Next you will be asked for the name of the file where you wish to save the compiled version of your program:

```
TO WHAT CODEFILE?
```

If you simply press the RETURN key the command will not be terminated, as you might expect. Instead, the compiled version of your program will be saved on the boot diskette's workfile SYSTEM.WRK.CODE . This is handy if you then wish to Run the program. Pressing the ESC key in response to this prompt returns you to the outermost Commmand level.

If you want the compiled version of your program to have the same name as the text version of your program (of course, the suffix will be .CODE instead of .TEXT), just type a dollar sign ($) and press the RETURN key. This is a handy feature, since you will usually want to remember only one name for both versions of your program. The dollar sign repeats your entire source file specification, including the volume identifier, so do NOT specify the diskette before typing the dollar sign. Note that this use is different from the use of the dollar sign in the Filer.

If you want your program stored under another filename, type the desired filename.

It is not necessary to type the suffix .CODE ; that suffix is automatically supplied by the Compiler. If you wish to defeat this feature, in order to specify an output filename that does not have a .CODE suffix, type a period (.) after the last character of your output filename. Ending your output filename with a file size specification (with or without a following period) also suppresses the addition of the .CODE suffix. The file is then opened observing the given file size (on closing, the file will have its actual size). The default file size for opening this file is [0]. See this manual's chapter THE FILER for details about file size specifications.

The Compiler then generates P-code codefiles to run directly on the Pascal P-machine.

When the Apple is compiling, messages on the screen show the progress of the compilation. The Apple Pascal Language Reference Manual describes these messages in detail.

If the compilation is successful (that is, no programming errors are detected), the Compiler saves the compiled code under the filename SYSTEM.WRK.CODE on the boot diskette, or under another filename that you specified earlier.

The code workfile, SYSTEM.WRK.CODE, is automatically erased when any text workfile is U(pdated from the Editor.

Should the Compiler detect an error in your program, the screen will show the text surrounding the error, an error number, and a marker <<<< pointing to the symbol in the source where the error was detected. For example, you might see the following message:

```
[ <<<<
LINE 9, ERROR 18: <SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

Pressing the spacebar instructs the Compiler to continue the compilation, in case you want to find more of the errors right now. Pressing the ESC key causes termination of the compilation and return to the Command level. Typing E sends you to the Editor, which automatically reads in the workfile, ready for editing.

Any error whose number is 400 or greater signifies a fatal error which causes ccompilation to terminate even if the spacebar is pressed.

If you were compiling a file that was not the workfile, this prompt appears when you enter the Editor:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
```

You should respond by typing the filename of the file you were compiling, and that file will then be read into the Editor. Only the main program file will be read into the Editor automatically. If your program uses Include-files, those files must be read in separately. See the Apple Pascal Language Reference Manual for details about the Include-file compiler option.

When the correct file has been read into the Editor, the first line displays the error message (or the error number, if the file SYSTEM.SYNTAX was not available on the boot diskette) and the cursor is placed at the symbol where the error was detected.

The error messages corresponding to the error numbers reported by the Apple Pascal Compiler are given the Apple Pascal Language Reference Manual. Some people may wish to save room on the boot diskette by removing SYSTEM.SYNTAX and using the printed table of messages, instead.

The use of compiler control options is described in the Apple Pascal Language Reference Manual.

CHAPTER 6

THE 6502 ASSEMBLER

136 INTRODUCTION
136 Diskfiles Needed
138 USING THE ASSEMBLER
142 Reference Symbol Table
142 Example
143 An Assembly-Language Routine
144 The Assembled Output Listing
149 A Pascal Program which Calls the Assembled Routine
149 Compiling, Linking and Running the Calling Program
151 ASSEMBLER INFORMATION
151 Syntax of Assembly Files
152 Syntax of Assembly Statements
152 Identifiers
152 Labels
152 Local Labels
153 Operators
153 Constants
153 Expressions
155 Linkage to Assembly Routines
157 THE ASSEMBLER DIRECTIVES
157 An Overview
158 Routine-Delimiting Directives
.PROC .FUNC .END
159 Label Definitions and Space Allocation Directives
.ASCII .BLOCK .EQU .ABSOLUTE
.BYTE .WORD .ORG .INTERP
162 Macro Facility Directives
.MACRO .ENDM
165 Conditional Assembly Directives
.IF .ELSE .ENDC
166 Host-Communication Directives
.CONST .PUBLIC .PRIVATE
167 External Reference Directives
.DEF .REF
169 Listing Control Directives
.LIST .MACROLIST .PATCHLIST .PAGE
.NOLIST .NOMACROLIST .NOPATCHLIST .TITLE
172 File Directive
.INCLUDE

172 ASSEMBLER DIRECTIVE SUMMARY
172 Metasymbol Notation
172 Routine Delimiting Directives
173 Label Definitions and Space-Allocation Directives
173 Macro Facility Directives
173 Conditional Assembly Directives
173 Host-Communication Directives
173 External Reference Directives
174 Listing Control Directives
174 File Directive

INTRODUCTION

You may occasionally wish to write small assembly-language routines and use them within a Pascal host program to provide low-level or time-critical facilities. The Assembler (in conjunction with the Linker) meets this need. Apple Pascal's 6502 Assembler is a version of the UCSD Adaptable Assembler, specifically implemented for the 6502 microprocessor on which the Apple Computer is based.

This assembler was modeled after The Last Assembler (TLA) developed at the University of Waterloo. The basic concept behind both the TLA and the UCSD Adaptable Assemblers is the use of a central machine-independent core that is common to all versions of the assembler. This central core is augmented with machine-specific code to handle the peculiarities of each individual machine.

This chapter is intended for a reader who is already fluent in at least one assembly language.

DISKFILES NEEDED

The following diskfiles allow you to use the Apple Pascal system's 6502 Assembler:

SYSTEM.ASSMBLER	(any diskette, any drive; required)
6500.OPCODES	(any diskette, any drive; required)
6500.ERRORS	(any diskette, any drive; optional error messages given in Assembler)
Textfile to be Assembled	(any diskette, any drive; default is boot diskette's workfile SYSTEM.WRK.TEXT, any drive)
SYSTEM.EDITOR	(any diskette, any drive; optional; to fix errors found by Assembler)

SYSTEM.ASSMBLER contains the Assembler itself, and 6500.OPCODES contains the op codes for the 6500 series of microprocessors (the Apple uses a 6502). These files are normally found on diskette APPLE2: . They must be available on any diskette in any of the system's disk drives when you type A from Command level to invoke the Assembler.

6500.ERRORS , normally found on diskette APPLE2: , is an optional file containing the Assembler syntax error messages. If it is not available, the Assembler will report syntax errors by number and you can look up the error description in Appendix D, Table 6.

When an assembly error is detected, you are given the option of returning directly to the Editor to correct the problem. If you type

E for E(dit from the Assembler, the file SYSTEM.EDITOR (normally found on your system diskette, APPLE0: or APPLE1:) should be available at that time, on any diskette in any disk drive.

One-drive note: In order to edit and assemble a textfile on a one-drive system, you may wish to use the Filer to T(ransfer the files SYSTEM.ASSMBLER and 6500.OPCODES from APPLE2: to APPLE1: , and then use APPLE1: as your boot diskette. This leaves 30 blocks unused on APPLE1: , for your source textfile and workfiles.

Multi-drive note: With APPLE1: in the boot drive and APPLE2: in drive volume #5:, your system has all the files needed to E(dit, C(ompile, A(ssemble, and L(ink programs.

Two-drive note: If you wish to E(dit and A(ssemble a textfile that is not already on APPLE1: or APPLE2: , and your system has only two drives, use the Filer's T(ransfer command to transfer that textfile onto either APPLE1: or APPLE2: before assembling.

A hidden disk need is the Assembler's use of a small area (usually less than four blocks) on the boot diskette, in any drive, to store a temporary intermediate file containing Linker information. This diskette file does not normally appear in the diskette's directory, but space for it must be available on the boot diskette.

An attempt to assemble without boot diskette space for this intermediate file (after opening both the output codefile and the assembled listing's output textfile) causes the message IO ERROR: NO ROOM ON VOL , after which you must press the spacebar to reinitialize the system. Your boot diskette may then show a new file named LINKER.INFO, of zero length and type "Infofile", often between two existing files. You may remove this file or not, as you wish.

In ordinary use of the Assembler, this problem does not arise, as the file size specification [*] is the default used when opening both the output codefile and the assembled listing's textfile. This default--unusual in the Apple Pascal system--automatically saves room for the Assembler's intermediate file.

However, if the output codefile exceeds the default file size, that file is automatically extended to maximum [0] size. If there was only one unused area on the boot diskette, this extension will eliminate the space needed by the temporary file. You can overcome this unlikely problem by specifying an appropriate file size for the output codefile, or by making sure there are at least two non-contiguous unused areas on the diskette.

If an assembly is so large that the Apple's available memory is insufficient, you can use the Filer to M(ake a four block file called SYSTEM.SWAPDISK on the diskette containing SYSTEM.ASSEMBLER . Before you attempt to read a new diskette directory (when finding an Include-file, for example), the system will first temporarily store the bottom

2K bytes of the data Heap in SYSTEM.SWAPDISK and then put the diskette directory into that 2K area of the Heap. This saves 2K bytes of top-of-Heap memory. When the directory is no longer needed, the contents of SYSTEM.SWAPDISK are stored back into their original Heap locations.

USING THE ASSEMBLER

The Assembler is invoked by typing A for A(ssemble from the outermost Command level of the Apple Pascal operating system. The screen immediately shows the message

```
          ASSEMBLING...
```

The Assembler automatically assembles the boot diskette's workfile SYSTEM.WRK.TEXT or another workfile designated by the Filer's G(et command and saves the resulting code (if assembly is successful) as SYSTEM.WRK.CODE . If no workfile is available, you are prompted for a source filename:

```
          ASSEMBLE WHAT TEXT?
```

You should respond by typing the name of the text file that you wish to have assembled.

It is not necessary to type the suffix .TEXT ; that suffix is automatically supplied by the Assembler, if you don't type it. If you wish to defeat this suffix-adding feature, to assemble a textfile whose filename does not end in .TEXT , type a period (.) after the last character of your filename.

Next you will be asked for the name of the file where you wish to save the assembled version of your routine:

```
          TO WHAT CODEFILE?
```

Pressing ESC in response to this prompt returns you to the outermost Command level.

If you simply press the RETURN key the command will not be terminated, as you might expect. Instead, the assembled version of your routine will be saved on the boot diskette's workfile SYSTEM.WRK.CODE .

If you want the assembled version of your routine to have the same name as the text version of your routine (of course, the suffix will be .CODE instead of .TEXT), just type a dollar sign ($) and press the RETURN key. This is a handy feature, since you will usually want to remember only one name for both versions of your routine. The dollar sign repeats your entire source file specification, including the volume identifier, so do NOT specify the diskette before typing the dollar sign.

If you want your routine stored under another filename, type the desired filename.

It is not necessary to type the suffix .CODE ; that suffix is automatically supplied by the Assembler. If you wish to defeat this suffix-adding feature, in order to specify an output filename that does not have a .CODE suffix, type a period (.) after the last character of your output filename.

Ending your output filename with a file size specification (with or without a following period) also suppresses the addition of any suffix. The file is then opened observing the given file size (on closing the file will have its actual size).

The usual default file size for opening this file is [*]. If this default allocates insufficient space for your codefile, you may wish to specify a different file size. Ending the given filename with a period changes the default file size to [0]. If the output codefile is being stored on the boot diskette, and if there is only one unused space on the diskette, this causes the output codefile to initially occupy all remaining space on the diskette, leaving no room for the required assembly-intermediate file.

Now that the source and object files for the assembly have been specified, the next prompt line is:

```
6500   ASSEMBLER  II.0   [D.4]
OUTPUT FILE FOR THE ASSEMBLED LISTING (<CR> FOR NONE):
```

You may now specify where you want the Assembler to send the assembled listing, or just press the RETURN key if you do not want this listing. (If you wish to abandon the assembly at this point, just press the ESC key.) If you specify a diskfile for the assembled listing, you do not need to type the .TEXT suffix; .TEXT will be added for you automatically if it is needed. Unlike many parts of the system, ending the specified filename with a period does NOT suppress the addition of the .TEXT suffix. However, if the filename you type includes the string .TEXT anywhere in it, the filename is used exactly as typed. If the filename ends in a .TEXT followed by a file size specification, the file is opened observing the given file size (on closing the file will have its actual size). The default size for opening this file is [*].

The assembled listing is a detailed display of the progress of the assembly showing location numbers, object code, source code, and other useful information. This listing is independent of the minimal assembled object code that is saved as the final output of the assembly. If an assembled listing is stored as a diskette file, it can be sent to the Editor, but non-standard control characters in the file make it very difficult to edit. For instance, form feeds (CTR-Ls) in the file are interpreted as clear screen characters by the Editor. Type /R/<CTRL>//<RET>/ immediately after entering the Editor for easier reading. See the EXAMPLE later in this chapter for a sample assembled listing.

If you wish, you can have the assembled listing sent to a diskette file or to the screen or printer. As usual for a console or printer output, the words CONSOLE or PRINTER must be followed by a colon, i.e. CONSOLE: . If the colon is omitted the listing is sent to a file of the name given, on the Prefix diskette. At this point, the program reports whether or not the output device (if any) is on line.

The program then starts assembling the workfile. If you did not tell the Assembler to send the assembled listing to CONSOLE: , a simple display of the assembly's progress appears on the screen. As assembly of your routine continues, the Assembler displays, on the left-hand side of the screen, one dot for each line of code assembled and a line counter every 50 lines. Upon completing each procedure or function, you will see the number of words of available symbol table space in brackets and the message

```
CURRENT MINIMUM SPACE IS xx WORDS
```

When an Include-file is started, the Assembler displays on the screen:

```
.INCLUDE <filename>
```

indicating which file has been included. If you told the Assembler to send the assembled listing to CONSOLE: , that listing replaces the simpler screen display.

If the Assembler encounters an error, a message shows the offending text and indicates the nature of the error. For example, you might see

```
    $04  .EQU $
IDENTIFIER PREVIOUSLY DECLARED
```

The error message will be taken from the file 6500.ERRORS if possible. If that is not possible, due to space limitations or the absence of the errors file, the error message number is given. In that case, you might see

```
    $04  .EQU $
ERROR #   9
"6500.ERRORS" FILE NOT AROUND
```

A complete list of Assembler syntax error messages corresponding to these error numbers appears in this manual's TABLES appendix. Note that the descriptive error message is given at the time the error is detected, not on entering the Editor as it is done in the Compiler. After each error is found, you are given the following choice:

```
E(DIT,<SPACE>,<ESC>
```

This is similar to the choice that you are given when the Compiler encounters an error. If you wish to proceed with the assembly, looking for more errors, press the spacebar. If you just wish to terminate the assembly, returning to the outermost Command level, press the ESC key. If you type E , the Editor is loaded into the computer, and the workfile is read into the Editor, ready for editing. If you were assembling a file which was not the workfile, this prompt appears:

```
>EDIT:
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE <ESC-RET> TO EXIT )
```

You should now type the filename of the sourcefile used for the assembly or the name of the Include-file if one exists. That file will then be read into the Editor.

When the correct file has been read in to the Editor, the first line displays an error message (or an error number if the file SYSTEM.SYNTAX is not available on the boot diskette) and the cursor is placed at the point in the text where the error was detected.

The Editor does not display specific error messages reported by the Assembler. Therefore before entering the Editor from the Assembler you should first note the specific error reported by the Assembler.

At the end of a completed assembly, the Assembler indicates that it is finished and tells you how many errors were found.

If the assembly was successful, the assembled code is written out to the boot diskette's workfile SYSTEM.WRK.CODE or (if no workfile was available) to the file that you specified in the beginning. This code file cannot be executed by itself but must be used by Linking it in with a Pascal host program file. For information about manual Linking, see the EXAMPLE later in this chapter, and also see this manual's chapter THE LINKER. For information about placing an assembled routine into SYSTEM.LIBRARY , so that R(un will automatically link the correct library routines into the Pascal host program, see this manual's chapter UTILITY PROGRAMS.

The boot diskette's code workfile, SYSTEM.WRK.CODE, is automatically erased when any text workfile is U(pdated from the Editor.

REFERENCE SYMBOL TABLE

In the assembly listing, an alphabetic reference symbol table (SYMBOLTABLE DUMP) is generated following the assembly of each procedure or function. Each entry in the reference symbol table is divided into three parts. The first part is the symbol identifier, the second part shows the symbol type, and the third part shows the value (if the symbol represents an absolute) or the definition location (if the symbol represents a label). The definition location is given as a high-byte first number and corresponds to one of the index numbers in the left-most column of the assembly listing. If the symbol represents neither an absolute nor a label, dashes appear in the third part of the entry. A vertical bar (|) ends each entry. Here is a reference symbol table from the upcoming example:

```
PAGE - 2       PADDLE      FILE:ASMDEMO     SYMBOLTABLE DUMP

AB - Absolute     LB - Label      UD - Undefined      MC - Macro
RF - Ref          DF - Def        PR - Proc           FC - Func
PB - Public       PV - Private    CS - Consts

DONE      LB 001F|  PADDLE    FC ----|  POP      MC ----|  PREAD2
          LB 0016|  RETURN    AB 0000|
```

The first entry shows a LaBel named DONE, defined at location 001F. The second entry shows that PADDLE is the name of the FunCtion. The last entry shows that RETURN is an ABsolute which has been assigned the value 0000 . Note that the fourth entry

```
PREAD2    LB 0016|
```

is broken onto two different lines.

EXAMPLE

This example will show the following:

1) A sample assembly-language routine which includes an external function (.FUNC), and an external procedure (.PROC).

2) The assembled routine, showing the Assembler's complete output listing.

3) A Pascal program which calls our assembly-language external function and procedure.

4) How to Compile the Pascal program and then Link in the assembly-language routine, in order to eXecute the program.

An Assembly-Language Routine

The following sample assembly-language routine contains an external function and an external procedure. The function is the game paddle function, and the procedure is the routine to set or clear one of the game TTL outputs. Both are provided for you, completely assembled and ready to use, in the UNIT named APPLESTUFF (see the Apple Pascal Language Reference Manual). See the later sections of the current chapter for details about the Assembler directives .FUNC and .PROC .

The following shows the assembly-language routine, just as you might type it into the computer, using the Editor:

```
;-----------------------------------------------
;  SAMPLE MACRO POPS 16 BIT ARGUMENT
;
        .MACRO POP
        PLA
        STA %1
        PLA
        STA %1+1
        .ENDM


        .FUNC PADDLE,1   ;ONE WORD OF PARAMETERS
;-----------------------------------------------
; SAMPLE GAME PADDLE FUNCTION FOR PASCAL
; (This function provided in APPLESTUFF unit.)
;
; FUNCTION PADDLE(SELECT: INTEGER): INTEGER;
;-----------------------------------------------
RETURN  .EQU    0        ;TEMP VAR FOR RETURN ADDR
                         ;NOTE: 0..35 HEX AVAILABLE

        POP RETURN       ;SAVE PASCAL RETURN ADDR
        PLA              ;DISCARD 4 BYTES STACK BIAS
        PLA              ;( ONLY DO FOR .FUNC )
        PLA
        PLA
        PLA              ;GET LSB SELECT PARAMETER
        AND #3           ;FORCE INTO RANGE 0..3
        TAX
        PLA              ;DISCARD MSB SELECT PARAM
        LDA 0C070        ;TRIGGER PADDLES
        LDY #0           ;INIT COUNT IN Y REG
        NOP              ;COMPENSATE FIRST COUNT
        NOP
PREAD2  LDA 0C064,X      ;TEST PADDLE
        BPL DONE         ;BRANCH IF TIMER DONE
        INY              ;ELSE INC Y EVERY 12 USEC
        BNE PREAD2       ;LOOP UNLESS 255 EXCEEDED
        DEY              ;MAKE 0 INTO 255 (MAX COUNT)
```

```
DONE    LDA #0
        PHA             ;PUSH MSB OF RETURN VALUE=0
        TYA
        PHA             ;PUSH LSB OF RETURN VALUE
        LDA RETURN+1    ;RESTORE PASCAL RETURN ADDR
        PHA
        LDA RETURN
        PHA
        RTS             ;AND RETURN TO PASCAL CALLER


        .PROC TTLOUT,2  ;TWO WORDS OF PARAMETERS
;-------------------------------------------------
;  ROUTINE TO SET OR CLEAR ONE OF THE TTL I/O BITS
;  (This procedure provided in APPLESTUFF unit)
;
;  PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);
;-------------------------------------------------
RETURN  .EQU    0       ;TEMP RETURN ADDR

        POP RETURN      ;SAVE PASCAL RETURN ADDRESS
                        ;POP PARAMETERS, LAST FIRST
        PLA             ;GET LSB BOOLEAN DATA 1=TRUE
        LSR A           ;SAVE BOOLEAN IN CARRY
        PLA             ;DISCARD MSB BOOLEAN DATA
        PLA             ;GET LSB SELECT
        AND #03         ;TREAT IT MOD 4
        ROL A           ;DOUBLE, ADD DATA FOR INDEX
        TAY             ;PUT I/O STROBE INDEX IN Y
        LDA 0C058,Y     ;ACTIVATE I/O STROBE
        PLA             ;DISCARD MSB SELECT PARAM
        LDA RETURN+1    ;RESTORE PASCAL RETURN ADDR
        PHA
        LDA RETURN
        PHA
        RTS             ;GO BACK TO PASCAL

        .END            ;END OF ASSEMBLY
```

The Assembled Output Listing

The preceding assembly-language routine, which we saved in the diskette file named MYDISK:ASMDEMO.TEXT , can now be assembled by the Apple Pascal 6502 Assembler. From the Command level, type the letter A for A(ssemble, and a dialog similar to the following takes place:

```
ASSEMBLING...
ASSEMBLE WHAT TEXT?  MYDISK:ASMDEMO
TO WHAT CODE FILE?  MYDISK:ASMDEMO
6500   ASSEMBLER  II.0 [D.4]
OUTPUT FILE FOR ASSEMBLED LISTING: (<CR> FOR NONE)  PRINTER:
```

The first response tells the Assembler to take the text version of the routine from the file MYDISK:ASMDEMO.TEXT . The second response says to save the assembled code version of the routine (when the assembly is complete) in the file MYDISK:ASMDEMO.CODE . By using the same name, we have fewer filenames to remember, and most commands will automatically choose the correct version (text or code). You could have accomplished the same thing by typing a dollar sign ($) as the second response.

If the text version of the routine had been available in the boot diskette's workfile SYSTEM.WRK.TEXT (or another workfile designated by the Filer's G(et command), the Assembler would automatically have assembled that file and would automatically have stored the assembled code version as SYSTEM.WRK.CODE .

The last response (PRINTER:) sends the assembled listing to the printer, and lets the usual Assembler display appear on the screen. The screen display looks something like this:

```
[11038]<    0>.............
2 BLOCKS FOR PROCEDURE CODE  9834 WORDS LEFT
[ 9827]<   13>.................................
[ 9960]<   50>........
CURRENT MINIMUM SPACE IS 9794 WORDS
[ 9787]<   58>...............................
CURRENT MINIMUM SPACE IS 9794 WORDS
[ 9808]<   90>
ASSEMBLY COMPLETE:          90 LINES
     0   ERRORS FLAGGED ON THIS ASSEMBLY
```

Meanwhile, the printer has been printing the assembled listing (if we had responded CONSOLE: , the assembled listing would have replaced the Assembler's usual screen display). The assembled listing appears approximately as shown below:

```
-----------------------------------------------------------------
PAGE - 0

Current memory available:   10613

0000|
0000|                   ;---------------------------------------
0000|                   ;
0000|                   ; SAMPLE MACRO POPS 16 BIT ARGUMENT
0000|                   ;
0000|                           .MACRO POP
0000|                           PLA
0000|                           STA %1
0000|                           PLA
0000|                           STA %1+1
0000|                           .ENDM
0000|
0000|

 2 blocks for procedure code  9834 words left
```

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PAGE - 1     PADDLE     FILE:ASMDEMO

0000|                         .FUNC PADDLE,1  ;ONE WORD OF PARAMETERS

Current memory available:     10056

0000|                   ;-------------------------------------
0000|                   ;
0000|                   ; SAMPLE GAME PADDLE FUNCTION FOR PASCAL
0000|                   ; (This function provided in APPLESTUFF unit.)
0000|                   ;
0000|                   ; FUNCTION PADDLE(SELECT: INTEGER): INTEGER;
0000|                   ;
0000|                   ;-------------------------------------
0000| 0000              RETURN  .EQU    0         ;TEMP VAR FOR RETURN ADDR
0000|                                             ;note: 0..35 hex available
0000|
0000|                           POP RETURN        ;SAVE PASCAL RETURN ADDR
0000| 68                #       PLA
0001| 85 00             #       STA RETURN
0003| 68                #       PLA
0004| 85 01             #       STA RETURN+1
0006| 68                        PLA               ;DISCARD 4 BYTES STACK BIAS
0007| 68                        PLA               ;( ONLY DO FOR .FUNC )
0008| 68                        PLA
0009| 68                        PLA
000A| 68                        PLA               ;GET LSB SELECT PARAMETER
000B| 29 03                     AND #3            ;FORCE INTO RANGE 0..3
000D| AA                        TAX
000E| 68                        PLA               ;DISCARD MSB SELECT PARAM
000F| AD 70C0                   LDA 0C070         ;TRIGGER PADDLES
0012| A0 00                     LDY #0            ;INIT COUNT IN Y REG
0014| EA                        NOP               ;COMPENSATE FIRST COUNT
0015| EA                        NOP
0016| BD 64C0           PREAD2  LDA 0C064,X       ;TEST PADDLE
0019| 10**                      BPL DONE          ;BRANCH IF TIMER DONE
001B| C8                        INY               ;ELSE INC Y EVERY 12 USEC
001C| D0F8                      BNE PREAD2        ;LOOP UNLESS 255 EXCEEDED
001E| 88                        DEY               ;MAKE 0 INTO 255 (MAX COUNT)
0019* 00
001F| A9 00             DONE    LDA #0
0021| 48                        PHA               ;PUSH MSB OF RETURN VALUE=0
0022| 98                        TYA
0023| 48                        PHA               ;PUSH LSB OF RETURN VALUE
0024| A5 01                     LDA RETURN+1      ;RESTORE PASCAL RETURN ADDR
0026| 48                        PHA
0027| A5 00                     LDA RETURN
0029| 48                        PHA
002A| 60                        RTS               ;AND RETURN TO PASCAL CALLER
002B|
002B|
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PAGE - 2     PADDLE     FILE:ASMDEMO     SYMBOLTABLE DUMP

AB - Absolute     LB - Label     UD - Undefined     MC - Macro
RF - Ref          DF - Def       PR - Proc          FC - Func
PB - Public       PV - Private   CS - Consts

DONE      LB 001F|  PADDLE     FC ----|  POP      MC ----|
PREAD2    LB 0016|  RETURN     AB 0000|


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PAGE - 3     PADDLE     FILE:ASMDEMO

Current minimum space is 9794 words


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PAGE - 4     TTLOUT     FILE:ASMDEMO

0000|                            .PROC TTLOUT,2  ;TWO WORDS OF PARAMETERS

Current memory available:   10056

0000|               ;----------------------------------------------
0000|               ;
0000|               ;  ROUTINE TO SET OR CLEAR ONE OF THE TTL I/O BITS
0000|               ;  (This procedure provided in APPLESTUFF unit)
0000|               ;
0000|               ;  PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);
0000|               ;
0000| 0000          RETURN  .EQU    0       ;TEMP RETURN ADDR
0000|
0000|                       POP RETURN      ;SAVE PASCAL RETURN ADDRESS
0000| 68           #        PLA
0001| 85 00        #        STA RETURN
0003| 68           #        PLA
0004| 85 01        #        STA RETURN+1
0006|                                       ;POP PARAMETERS, LAST FIRST
0006| 68                    PLA             ;GET LSB BOOLEAN DATA 1=TRUE
0007| 4A                    LSR A           ;SAVE BOOLEAN IN CARRY
0008| 68                    PLA             ;DISCARD MSB BOOLEAN DATA
0009| 68                    PLA             ;GET LSB SELECT
000A| 29 03                 AND #03         ;TREAT IT MOD 4
000C| 2A                    ROL A           ;DOUBLE, ADD DATA FOR INDEX
000D| A8                    TAY             ;PUT I/O STROBE INDEX IN Y
000E| B9 58C0               LDA 0C058,Y     ;ACTIVATE I/O STROBE
0011| 68                    PLA             ;DISCARD MSB SELECT PARAM
0012| A5 01                 LDA RETURN+1    ;RESTORE PASCAL RETURN ADDR
0014| 48                    PHA
0015| A5 00                 LDA RETURN
0017| 48                    PHA
```

```
0018| 60                    RTS              ;GO BACK TO PASCAL
0019|
0019|
0019|
0019|                         .END             ;END OF ASSEMBLY

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PAGE - 5     TTLOUT     FILE:ASMDEMO     SYMBOLTABLE DUMP

AB - Absolute     LB - Label     UD - Undefined     MC - Macro
RF - Ref          DF - Def       PR - Proc          FC - Func
PB - Public       PV - Private   CS - Consts


POP       MC ----| RETURN     AB 0000| TTLOUT     PR ----|


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
PAGE - 6     TTLOUT     FILE:ASMDEMO

Current minimum space is 9794 words

Assembly complete:         92 lines

     0   Errors flagged on this Assembly

----------------------------------------------------------------------
```

Only the assembled object code from the preceding assembly listing (the second column of information on PAGE-1 and on PAGE-4) is saved in the file MYDISK:ASMDEMO.CODE .

NOTES about the preceding sample assembly listing:

1) The location values in the symbol table dump refer to the locations in the listing.

2) The ** 's in the listing (see PAGE-1 of the listing, for example) call attention to the use of a label not yet defined. The Assembler displays one * for each hexadecimal digit to be filled in later.

3) If a * appears after the location number at the left of the listing, it indicates that a forward reference occurring earlier in the assembly has been resolved. The number to the left of the * is the location where the reference occurred while the number to the right is the new contents of that location. See PAGE-1 of the listing, for example.

4) The Apple Pascal 6502 Assembler uses the following non-standard notation for indirect addressing:

Apple Pascal 6502 Assembler	Standard 6502 Assembler
LDA @LOC1,Y	LDA (LOC1),Y
LDA @LOC2,X	LDA (LOC2,X)
JMP @GOVECT	JMP (GOVECT)

A Pascal Program which Calls the Assembled Routine

The following is a sample Pascal program which uses the external function and procedure assembled earlier:

```
PROGRAM CALLASM;

(* DEMONSTRATES CALLING ASSEMBLY LANGUAGE PROCEDURES *)

VAR I: INTEGER;

PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);
EXTERNAL;

FUNCTION PADDLE(SELECT: INTEGER): INTEGER;
EXTERNAL;

BEGIN
  FOR I:= 1 TO 1000 DO
    BEGIN
      WRITELN(PADDLE(0):3,'    ',PADDLE(1):3);
      TTLOUT(0,ODD(I))
    END
END.
```

Compiling, Linking and Running the Calling Program

To use the Pascal program CALLASM, you must compile the text version shown above to make a compiled P-code version. This is done from the Command level by typing C for C(ompile. When you do this, a dialog similar to the following takes place (our program CALLASM was saved in the text file named MYDISK:CALLASM.TEXT):

```
COMPILING...
COMPILE WHAT TEXT?  MYDISK:CALLASM
TO WHAT CODEFILE?  MYDISK:CALLASM
```

The first response tells the Compiler to compile the text that is found in the file MYDISK:CALLASM.TEXT . The second response tells the Compiler to save the resulting compiled code in the file MYDISK:CALLASM.TEXT . Again, we used the same name for the text version and for the code version of the program, to save us remembering two different names. (Again, we could have typed the second response as $.)

If the text version of the Pascal program had been available in the boot diskette's workfile SYSTEM.WRK.TEXT (or another workfile designated by the Filer's G(et command), the Compiler would automatically have compiled that text, and would have saved the resulting code version as SYSTEM.WRK.CODE on the boot diskette.

At this point, if there are no errors in the program, CALLASM is compiled; the resulting P-code version is stored as MYDISK:CALLASM.CODE . Messages similar to the following will then appear on the screen:

```
PASCAL COMPILER II.1 [B2B]
<   0>........
TTLOUT    [ 2533 WORDS]
<   8>...
PADDLE    [ 2504 WORDS]
<  11>...
CALLASM   [ 2515 WORDS]
<  14>.....
19 LINES
SMALLEST AVAILABLE SPACE = 2515 WORDS
```

However, CALLASM is still not ready to be run: the external assembly-language function and procedure in MYDISK:ASMDEMO.CODE must now be Linked into the Pascal program. To do this, from the Command level type an L for L(ink, and a dialog similar to the following will take place:

```
LINKING...

LINKER II.1  [A4]
HOST FILE?  MYDISK:CALLASM                  (Your main Pascal program)
OPENING MYDISK:CALLASM.CODE
LIB FILE?  MYDISK:ASMDEMO                      (The routine to link in)
OPENING MYDISK:ASMDEMO.CODE
LIB FILE?                                (Press RETURN for last file)
MAP NAME?                               (Press RETURN for no map file)
READING CALLASM
READING PADDLE
OUTPUT FILE?  MYDISK:FINALCALL.CODE    (Final product; note .CODE)
LINKING CALLASM  #  1
   COPYING FUNC PADDLE
   COPYING PROC TTLOUT
```

At last! The file FINALCALL.CODE now contains your compiled Pascal program CALLASM together with the linked-in assembly-language routines PADDLE and TTLOUT. You may now X(ecute the program FINALCALL.

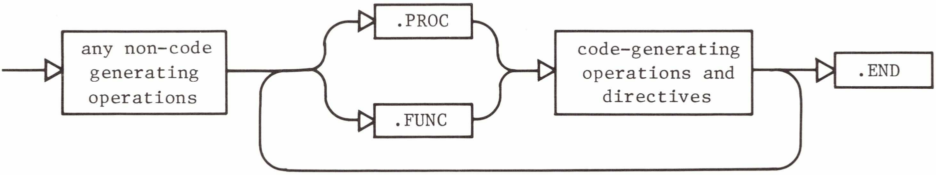
ASSEMBLER INFORMATION

SYNTAX OF ASSEMBLY FILES

All objects declared before the first .PROC or .FUNC are available for use throughout the assembly. No code is allowed to be generated before the first .PROC or .FUNC . The symbol table is reduced at the beginning of each .PROC or .FUNC to the point where it was at the start of the first .PROC or .FUNC .

All assemblies must end with a .END . However, each .PROC or .FUNC before the last one is ended by the occurrence of the next .PROC or .FUNC . Only the last one should end with a .END , as all text beyond the .END is ignored by the Assembler.

A general syntax diagram for all assembly files looks like this:


any non-code generating operations
.PROC
.FUNC
code-generating operations and directives
.END

ASSEMBLY-FILE SYNTAX

The non-code-generating directives are:

.EQU
.ABSOLUTE
.INTERP
.MACRO
.ENDM
.IF
.ELSE
.ENDC
.DEF
.REF
.LIST
.NOLIST
.MACROLIST
.NOMACROLIST
.PATCHLIST
.NOPATCHLIST
.PAGE
.TITLE

The body of a macro definition is a non-code-generating operation.

SYNTAX OF ASSEMBLY STATEMENTS

Identifiers

Identifiers are character strings starting with an alpha character. Other characters must be alphanumeric or the ASCII underline (_). Only the first eight characters are meaningful to the Assembler even though more may be entered.

The Assembler makes only one pass through the source. On encountering an undefined identifier in an expression, something must be assumed about the nature of the identifier in order for the assembly to continue. It is therefore assumed that the undefined identifier will eventually be defined as a label, which is the most probable case. Any identifier which is not a label must be defined before it is used.

Labels

Only labels and comments may begin in the first column, with no preceding spaces. Labels may optionally be followed by a colon. If a statement has no label, the first column must contain a space.

Labels may be equated to an expression containing labels and/or absolutes. You must define a label before it is used unless it will simply be equated to another label.

Local Labels

Local labels must have $ as the first non-space character, and may be up to 8 digits long. Local labels may not occur on the left-hand side of an equate (.EQU).

Local labels are mainly used to jump around within a small segment of code without having to use up storage area needed by regular labels. The local label stack may hold up to 21 labels. The local label stack is emptied each time a regular label is encountered, thus rendering the previous local labels invalid beyond that point in the assembly. An example of the use of local labels is shown below, where the jump to label $04 is made illegal by the intervening regular label REALLAB .

```
     $03    STA  4          ;LEGAL USE OF LOCAL LABEL
             .
             .
            BNE  $03
             .
             .
            BNE  $04        ;ILLEGAL USE OF LOCAL LABEL
 REALLAB    .EQU $
     $04    .EQU $
```

Operators

The following operators can be used in expressions processed by this Assembler:

For unary operations:

+	plus
-	minus
~	ones complement (not available on the Apple keyboard)

For binary operations:

+	plus
-	minus
^	exclusive or
*	multiplication
/	truncating division (DIV)
%	remainder division (MOD)
|	bit-wise OR (not available on the Apple keyboard)
&	bit-wise AND
=	equal (valid only in .IF)
<>	not equal (valid only in .IF)

All operators have the same precedence.

Constants

All constants must start with an integer from 0 through 9.
For example, the hexadecimal constant FF must be written 0FF

The default radix is Hexadecimal. Decimal constants must be followed by a period (.) .

EXAMPLE: Hexadecimal: 13
Decimal: 19.

Expressions

Expressions are evaluated by the Assembler from left to right, and all operators have the same precedence. To override the default, left-to-right precedence, use angle brackets <like this> .

A relocatable label can be used in an address expression such as

```
        LDA   RLABEL+5       ; Legal expression with label
```

but only if the expression ADDS or SUBTRACTS a constant value from the address of the label. An expression such as

```
        LDA   RLABEL*2       ; Illegal expression with label
```

will not be accepted by the Assembler unless you are assembling using the .ABSOLUTE directive (discussed later in this chapter) and RLABEL has previously been defined.

A relocatable label must not appear in an expression used to make an absolute constant. A statement such as

```
         LDA   #RLABEL+5      ; Illegal use of label as absolute constant
```

will not be accepted by the Assembler.

The following portion of an assembled listing illustrates expression syntax as used in the Assembler. The examples are not intended to form an actual, useful program.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 PAGE - 1      TEMP1      FILE:EXPRSYNTAX

0000|                                  .PROC TEMP1 ; SHOWS EXPRESSION SYNTAX
Current memory available:   10088
0000|                                              ; CONSTANTS
0000|
0000| 000A                    CON10    .EQU  10.
0000| 00BF                    OTH0     .EQU  0BFH
0000| 00F7                    ONE0     .EQU  0F7H
0000|
0000|                                              ; EXAMPLE EXPRESSIONS
0000|
0000| A5 05                             LDA  5
0002| A5 4D                             LDA  5+6*7
0004| A5 4D                             LDA  <5+6>*7
0006| A5 0A                             LDA  7*6/4
0008| A5 07                             LDA  7*<6/4>
000A| A5 01                             LDA  6%5
000C| A5 02                             LDA  5+11%5
000E| A5 07                   LABEL     LDA  5+<11%5>
0010| A5 48                             LDA  OTH0^ONE0
0012| A5 B7                             LDA  OTH0&ONE0
0014| AD 0E00                           LDA  LABEL

0017| AD 0900                           LDA  LABEL-5
001A| AD 4000                           LDA  LABEL+<5*CON10>

        LDA  LABEL*2
ill-formed expression
E(dit,<space>,<esc>       [ Spacebar pressed here, to continue assembly. ]

001D|                                   LDA  LABEL*2
001D| A9 05                             LDA  #5
001F| A9 1C                             LDA  #5*<CON10 / 2> +3

        LDA  #<LABEL+5>
operand not absolute
E(dit,<space>,<esc>       [ Spacebar pressed here, to continue assembly. ]
0021| A9                                LDA   #<LABEL+5>

        LDA   #LABEL
operand not absolute
E(dit,<space>,<esc>      [ Spacebar pressed here, to continue assembly. ]

0022| A9                                LDA   #LABEL
0023|
0023|                                  .END
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

LINKAGE TO ASSEMBLY ROUTINES

External assembly-language routines (.PROC's and .FUNC's) are separately assembled and often stored in a diskette library file such as the boot diskette's SYSTEM.LIBRARY . A Pascal host program that uses such external assembled routines must have those routines linked from their library file(s) into the compiled host program's codefile.

A Pascal host program declares that a routine is external in much the same way as a Pascal routine is declared FORWARD. A standard PROCEDURE or FUNCTION heading is provided, followed by the keyword EXTERNAL. Calls to the external routine use standard Pascal syntax, and the Compiler checks that each call to the external routine agrees in type and number of parameters with the original EXTERNAL declaration for that routine. It is the programmer's responsibility to assure that the assembly-language routine respects the Pascal EXTERNAL declaration. The Linker checks only that the number of words of parameters agree between the Pascal EXTERNAL declaration and the corresponding assembly-language .PROC or .FUNC declaration. For more information see this manual's chapter THE LINKER.

When the Pascal host program calls an external assembly-language routine, passed parameters are pushed on the evaluation stack as they are encountered in the host program's calling statement. The first parameter is pushed on the stack, highest byte first, then the second parameter, and so on. Long integers and sets are always passed as the maximum number of words allocated by the host program's long integer or set declaration, each word high byte first. A word indicating the length in words is pushed last. Strings, records, arrays, and VARiables are passed by address, pushing the high byte first, then low byte. The Pascal host program's EXTERNAL declaration may declare a VARiable without type. This allows a parameter of indeterminate size to be passed by address. When all the parameters have been passed, the host program's return address (the address to which the program must return on completing the external routine) is pushed on the stack, high byte first, then low byte.

In use, the assembly-language routine must save the return address, and must push it on the stack again just before returning to the calling program. The passed parameters are available on the stack in the reverse order to the order in which they were originally pushed on the stack.

The conventions of the surrounding system concerning register use and calling sequences must be respected by writers of assembly-language routines. On the Apple, all registers are available, and zero-page hexadecimal locations 0 through 35 are available as temporary variables. However, the Apple Pascal system also uses these locations as temporaries, so you should not expect data to remain there from one execution of a routine to the next. You can save variables in non-zero page memory by using the .BYTE or .WORD directives in your routine to reserve space.

For external assembly-language functions (.FUNC's) only, two additional conventions must be recognized:

1) At the function's entry time, the Pascal host program pushes two words (four bytes) of zeros on the evaluation stack after any passed parameters are put on the stack and before the return address is pushed on the stack.

2) At the function's exit time, the .FUNC must push the function result (a scalar, real, or pointer, maximum two words), high byte first, just before pushing the return address on the stack.

For an example of an external assembly-language procedure, an external assembly-language function, and a Pascal host program which calls these routines, see the EXAMPLE earlier in this chapter. The EXAMPLE also demonstrates the handling of the return address, passed parameters, and returned function value in assembly-language routines. The external routines in that example are manually Linked into the Pascal calling program. For information about installing a routine into the system library, see this manual's chapter UTILITY PROGRAMS.

THE ASSEMBLER DIRECTIVES

AN OVERVIEW

Assembler directives (also referred to as "pseudo-ops") let you tell the Assembler to do various functions other than provide directly executable code. The following directives are common to all versions of the UCSD Adaptable Assembler, including the Apple Pascal 6502 Assembler, but may differ from individual manufacturer′s standard syntax.

In the following descriptions of directives, square brackets [like this] are metasymbols that denote optional elements which you may supply. Angle brackets <like this> are meta-symbols that denote required elements which you must supply. If an element type is not shown, it cannot be used in that situation.

EXAMPLE:

```
              [label]    .ASCII    "<character string>"
```

This notation indicates that you may supply a label, but it is not necessary, and that between the required double quotes you must supply the character string to be converted (not necessarily the words "character string"). The bracket metasymbols are not to be be typed.

The following terms represent general concepts in the explanation of each directive:

TERM:	DEFINITION:
value	Any numerical value, label, constant, or expression.
valuelist	A list of one or more values separated by commas.
identifierlist	A list of one or more identifiers separated by commas.
expression	Any legal expression as defined under SYNTAX OF ASSEMBLY STATEMENTS.
identifier[:integer] list	A list of one or more identifier:integer pairs separated by commas. The colon-integer is optional in each pair and the default is 1.

Small examples are included after each directive definition to show you the specific syntax and form of that directive. The EXAMPLE assembly-language routine earlier in this chapter is used to show the combined use and detailed examples of directive operations.

ROUTINE-DELIMITING DIRECTIVES

Every assembly must include at least one .PROC or .FUNC, and one .END , even in the case of stand-alone code which will not be linked into a Pascal host (e.g., the interpreter). The most frequent use of the Assembler, however, will be small routines intended to be linked with a Pascal host. In this case, .PROCs and .FUNCs are used to identify and delimit the assembly code to be accessed by a Pascal external procedure or function. The .END appears at the end of the last routine and serves as the final delimiter.

References to an assembly-language .PROC or .FUNC are made in the Pascal host program by use of EXTERNAL declarations. At the time of this declaration the actual parameter names must be given. For example, if the Pascal host's declaration is:

```
PROCEDURE FARKLE(X,Y:REAL);
EXTERNAL;
```

the associated declaration for the assembly-language .PROC would be

```
          .PROC FARKLE,4
```

A .PROC, .FUNC, or any assembly routine should be inserted into the SYSTEM.LIBRARY so that it can be referenced by the Linker and linked into the Pascal host program at R(un time. An alternate method would be to execute the Linker and tell it what files to link in. Either method works. However, if the Pascal host is updated and the assembly routines have not been installed in the SYSTEM.LIBRARY, the Linker will have to be executed again after each host program update. Therefore, we suggest that the routines be inserted into the SYSTEM.LIBRARY to avoid this repetition. If the Linker is called automatically, using the R(un command, it will automatically search the SYSTEM.LIBRARY for the appropriate definition of the assembly routine and link the two together.

The EXAMPLE earlier in this chapter shows the use of assembly-language routines from a Pascal host program and demonstrates the manual linking process. More information on linking appears in this manual's chapter THE LINKER. For information on using the system librarian to install a routine into SYSTEM.LIBRARY , see this manual's chapter UTILITY PROGRAMS.

.PROC Identifies a procedure that returns no value. A .PROC is ended by the occurrence of a new .PROC , .FUNC , or .END .

FORM: `.PROC <identifier>[,expression]`

[expression] indicates the number of words of parameters expected by this routine. The default is 0.

EXAMPLE: `.PROC DLDRIVE,2`

.FUNC Identifies a function that returns a value. Two words of space to be used for the function value will be placed on the stack after any parameters. A .FUNC is ended by the occurrence of a new .PROC , .FUNC , or .END .

FORM: `.FUNC <identifier>[,expression]`

[expression] indicates the number of words of parameters expected by this routine. The default is 0.

EXAMPLE: `.FUNC RANDOM,4`

.END Used to denote the physical end of an assembly.

FORM: `.END`

EXAMPLE: `.END`

LABEL DEFINITIONS AND SPACE ALLOCATION DIRECTIVES

.ASCII Converts character values to ASCII equivalent byte constants and places the equivalents into the code stream.

FORM: `[label] .ASCII "<character string>"`

where <character string> is any string of printable ASCII characters, including a space. The length of the string must be less than 80 characters. The double quotes are used as delimiters for the characters to be converted. If a double quote is desired in the string, it must be specifically inserted using a .BYTE .

EXAMPLE: `.ASCII "HELLO"`

for the insertion of AB"CD the code must be constructed as:

```
.ASCII    "AB"
.BYTE      22     ; An ASCII  "
.ASCII    "CD"
```

Note: The 22 is the hexadecimal ASCII code for a double quote.

.BYTE Allocates a byte of space into the code stream for each value listed. Each value actually stored by the routine must have a value between -128 and +255. If the value is outside of this range an error will be flagged. Assigns the associated label, if any, to the address at which the byte was stored.

```
FORM:    [label]  .BYTE   [valuelist]

             the default for no stated value is 0.

EXAMPLE:    TEMP  .BYTE    4

             the associated output would be:  04
```

.BLOCK Allocates a block of space into the code stream for each value listed. Amount allocated is in bytes. Associates the label (if present) with the starting address of the block allocated.

```
FORM:    [label]  .BLOCK   <length>[,value]
```

<length> is the the number of bytes to hold the <value> specified. The default for no stated value is 0.

```
EXAMPLE:    TEMP  .BLOCK   4,6

             the associated output would be:

                 06
                 06         ( four bytes with the value 06 )
                 06
                 06
```

.WORD Allocates a word of space in the code stream for each value in the valuelist. Associates the declaration label with the word space allocation.

```
FORM:    [label]  .WORD   <valuelist>

EXAMPLE:    TEMP  .WORD   0,2,4,...

             the associated output would be:

                 0000
                 0002
                 0004   (words with these values in them)
                  .
                  .
```

```
EXAMPLE:    A1    .WORD  A2
                    .
                    .
                    .
            A2    .EQU   $          ;   $ denotes LC value
                  .WORD  5.
                    .
```

The statement `A2 .EQU $` assigns the current value of the location counter (LC) to the label A2 . If the value of the location counter is 50 at the .EQU , the associated output would be:

```
0050  ( assignment due to the value of L2 )
  .
  .
0005  ( assignment due to the .WORD 5 )
  .
```

.EQU Assigns a value to a label. Labels may be equated to an expression containing labels and/or absolutes. One must define a label before it is used unless it will simply be equated to another label. A local label may not appear on the left-hand side of an equate (.EQU).

```
FORM:     <label>  .EQU     <value>

EXAMPLE:   BASE    .EQU     R6
```

.ORG Takes the operand of .ORG as the offset, relative to the start of the assembly file, where the next word or byte of code is to go. Words or bytes of zeros are produced to get the current location counter (LC) to the correct value.

```
FORM:              .ORG     <value>

EXAMPLE:           .ORG     0D000
```

.ABSOLUTE If a .ABSOLUTE occurs before the first .PROC then all .ORG's are interpreted as absolute memory locations. The user must take responsibility for the correct loading of the produced code file. The use of .ABSOLUTE has the effect of cancelling the generation of relocation information. Further, any defined (i.e., non forward-referenced) labels may be treated as absolute numbers. Thus such labels may be multiplied and divided, etc. .ABSOLUTE must occur before the first .PROC and is set for the entire assembly.

```
FORM:              .ABSOLUTE

EXAMPLE:           .ABSOLUTE
```

.INTERP Interpreter relative locations are specified by the use of .INTERP in an expression. Further labels may defined as interpreter relative in the manner shown in the example. The rules regarding the use of such labels are the same as for any other specially defined labels (e.g., .PUBLIC and .PRIVATE). Locations whose values depend on interpreter relative labels or expressions are listed in a fourth relocation list at the end of the assembly procedure.

```
EXAMPLE:   STUFF  .EQU   .INTERP+25
```

Certain interpreter entry points may be useful, using an instruction such as

```
LDA   @.INTERP+n
```

with these values of n:

n=0	Address of the execution error routine; displays error message using the error number in the A register.
n=2	Address of the BIDS jump table; handles input and output.
n=4	Address of SYSCOM; system's communications area of the P-machine.

MACRO FACILITY DIRECTIVES

A macro is a named section of text that can be defined once and repeated in other places simply by using its name. The text of the macro may be parameterized, so that each invocation results in a different version of the macro contents. The entire macro definition may precede the first .PROC or .FUNC of the assembly file.

At the invocation point, the macro name is followed by a list of parameters, each terminated by a comma (except for the last one, which is terminated by end of line or the comment indication (;). The text of the macro definition, modified by substituting the invocation parameters, is inserted (conceptually speaking) by the Assembler at the invocation point. Wherever %n (where n is a single decimal digit greater that zero) occurs in the macro definition, the text of the n-th invocation parameter is substituted. Leading and trailing blanks are stripped from the parameter before the substitution. If the macro definition includes a reference to a parameter not provided in a particular invocation (too few parameters or no parameter before a terminating comma), a null string is substituted.

A macro definition may not contain another macro definition. A definition can certainly, however, include macro invocations. This "nesting" of macro invocations is limited to five levels deep.

The expanded macro is always included in the listing file (unless .NOMACROLIST is in effect at the point of invocation). Macro expansion text is flagged, in the listing, by a # just left of each expanded line. Comments occurring in the macro definition are not repeated in the expansion.

.MACRO	Indicates the start of a macro definition and gives it an identifier.
.ENDM	Indicates the end point of a macro definition.

```
FORM:              .MACRO     <identifier>
                     .
                     .                   ; (macro body)
                     .
                   .ENDM

EXAMPLE:           .MACRO HELP
                   STA   %1              ; < comment >
                   LDA   %2              ; < comment >
                   .ENDM
```

The assembly listing beginning at the point where this macro was invoked may look like this:

```
              HELP  ALPHA,BETA
        #     STA   ALPHA
        #     LDA   BETA
```

The statement HELP calls the defined macro and sends it two parameters, ALPHA and BETA. These parameters are in turn used in forming the macro expansion (flagged in the listing by # signs) that follows the invoking statement. In the expansion, the first calling-statement parameter (variable ALPHA) is substituted for the definition's identifier %1 , and the second parameter (variable BETA) is substituted for the identifier %2 .

The following portion of an assembled listing illustrates the syntax used when defining and invoking macros. The procedure itself is not meant to be an actual, useful program.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 PAGE -    1  TEMP2     FILE:MACROCALL

0000|                            .PROC TEMP2 ; SHOWS SYNTAX OF MACRO CALLS
Current memory available:   10088
0000|                                         ; CONSTANTS
0000|
0000| 000A              CON10    .EQU  10.
0000| 00BF              OTH0     .EQU  0BFH
0000| 00F7              ONE0     .EQU  0F7H
0000|
0000|                                         ; MACRO DEFINITIONS
0000|
0000|                            .MACRO M2
0000|                              CLC
0000|                              LDA   PREDEFL+%1
```

```
0000|                           .ENDM
0000|
0000|                           .MACRO TESTM
0000|                             JMP  %1
0000|                             LDA  #5+%2
0000|                             M2  %2 ; MACRO CALL WITHIN A MACRO DEF'N
0000|                             LDA  %3
0000|                             LDA  %4
0000|                             LDA  %5
0000|                             JMP  %6
0000|                           .ENDM
0000|
0000| A5 05           PREDEFL  LDA  5  ; A PRE-DEFINED LABEL
0002|
0002|                                   ; MACRO CALL WITH ALL PARAMETERS
0002|                                   ; & NO LEADING OR TRAILING SPACES
0002|
0002|                           TESTM PREDEFL,<5*CON10+6>,#55,#6,1,LABEL2
0002| 4C 0000            #       JMP  PREDEFL
0005| A9 3D              #       LDA  #5+<5*CON10+6>
0007|                    #       M2  <5*CON10+6>
0007| 18                 #       CLC
0008| AD 3800            #       LDA   PREDEFL+<5*CON10+6>
000B| A9 55              #       LDA  #55
000D| A9 06              #       LDA  #6
000F| A5 01              #       LDA  1
0011| 4C ****            #       JMP  LABEL2
0014|
0014|                           M2   5  ; SIMPLE MACRO CALL
0014| 18                 #       CLC
0015| AD 0500            #       LDA   PREDEFL+5
0018|
0018|                                   ; MACRO CALL WITH NUL PARAMETERS
0018|                                   ; AND LEADING & TRAILING SPACES
0018|
0018|                           TESTM  ,CON10,,   XX  ,0F0H, PREDEFL

       JMP
not enough operands
E(dit,<space>,<esc>          [ Spacebar pressed here, to continue assembly. ]

0018|                    #       JMP
0018| A9 0F              #       LDA  #5+CON10
001A|                    #       M2  CON10
001A| 18                 #       CLC
001B| AD 0A00            #       LDA   PREDEFL+CON10
```

```
        LDA
ill formed operand
E(dit,<space>,<esc>          [ Spacebar pressed here, to continue assembly. ]

001E|                         #       LDA
001E| AD ****                 #       LDA   XX
0021| A5 F0                   #       LDA   0F0H
0023| 4C 0000                 #       JMP   PREDEFL
0026|
0026|                               .END
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

CONDITIONAL ASSEMBLY DIRECTIVES

Conditionals are used to selectively exclude or include sections of code at assembly time. When the Assembler encounters a .IF directive, it evaluates the associated expression. In the simplest case, if the expression is false, the Assembler simply discards the text until a .ENDC is reached. If there is a .ELSE directive between the .IF and .ENDC directives, the text before the .ELSE is selected if the expression is true, and the text after the .ELSE if the condition is false. The unassembled part of the conditional will not be included in any listing. Conditionals may be nested.

The conditional expression takes one of two forms. The first is the normal arithmetic/logical expression used elsewhere in the Assembler. This type of expression is considered false if it evaluates to zero; true otherwise. The second form of conditional expression is comparison for equality (indicated by =) or inequality (indicated by <>). One may compare strings, characters, or arithmetic/logical expressions.

.IF	Identifies the beginning of the conditional.
.ENDC	Identifies the end of a conditional .IF
.ELSE	Identifies the alternate to the .IF If the conditional expression is equal to 0 then the else portion is used.

```
FORM:   [label]  .IF  <expression>
                   .
                   .
                   .
               [ .ELSE ]
                   .
                   .            ; (only if there is an else)
                   .
                 .ENDC
```

where the expression is the conditional expression to be met.

EXAMPLE:

```
                .IF LABEL1-LABEL2    ;Arithmetic expression.
                  .                  ; This text assembled
                  .                  ; only if subtraction
                  .                  ; result is non-zero
                  .

                .IF  "%1" ="STUFF"   ;Comparison expression.
                  .                  ; This text assembled
                  .                  ; if subtraction above
                  .                  ; was true and if text
                  .                  ; of first parameter
                  .                  ; (assume we're in macro)
                  .                  ; is equal to "STUFF".
                .ENDC                ; Terminates nested cond.
                  .
                  .
                .ELSE
                  .                  ; This text assembled if
                  .                  ; subtraction result was
                  .                  ; zero.
                  .
                .ENDC                ; Terminates outer level
                                     ; of conditional.
```

HOST-COMMUNICATION DIRECTIVES

The directives .CONST , .PUBLIC , and .PRIVATE allow the sharing of information and data space between an assembly routine and the host program which uses that routine. These external references must eventually be resolved by the Linker. Refer to this manual's chapter THE LINKER for further details.

.CONST Allows globally declared constants in the host program to be accessed by the assembly routine. .CONST can only be used in a program to replace 16-bit relocatable objects.

FORM: `.CONST    <identifierlist>`

EXAMPLE: (see example after .PRIVATE)

.PUBLIC Allows a variable declared in the global data segment of the host program to be used by both the assembly-language routine and the host program.

FORM: `.PUBLIC  <identifierlist>`

EXAMPLE: (see example after .PRIVATE)

.PRIVATE Allows variables of the assembly routine to be stored in the host program's global data segment and yet be inaccessible to the host program. These variables retain their values for the entire execution of the program.

FORM: .PRIVATE <identifier[:integer] list>

The integer is used to communicate the number of words to be allocated to the identifier. The default is one word.

EXAMPLE: (for .CONST, .PRIVATE, and .PUBLIC)

Given the following Pascal host program:

```
PROGRAM EXAMPLE;
CONST SETSIZE=50; LENGTH=80;

VAR I,J,F,HOLD,COUNTER,LDC:INTEGER;
    LST1:ARRAY[0..9] OF CHAR;

BEGIN
  .
  .
  .
END.
```

and the following section of an assembly routine:

```
.CONST     LENGTH
.PRIVATE   PRT,LST2:9
.PUBLIC    LDC,I,J
```

This will allow the constant LENGTH to be used in the assembly routine almost as if the line LENGTH .EQU 80. had been written. (Recall the limitation mentioned above for using .CONST identifiers.) The variables LDC,I and J are to be used by both the Pascal host and the assembly routine, while the variables PRT and LST2 are to be used only by the assembly routine. Further, the LST2:9 causes the variable LST2 to correspond with the beginning of a nine-word block of space in the Pascal host's global data segment.

EXTERNAL REFERENCE DIRECTIVES

Separate routines may share data structures and subroutines by linkage from one assembly routine to another assembly routine. This is made possible through the use of .DEF and .REF . These directives cause the Assembler to generate link information that allows two separately assembled routines to be linked together. By using .DEF and .REF , one assembly routine may call subroutines found in another assembly routine. One routine placed in a library file such as the boot diskette's SYSTEM.LIBRARY can contain a large number of frequently used subroutines which are all available to other routines.

The use of .DEF and .REF is similar to that of .PUBLIC . .DEFs and .REFs associate labels between two assembly routines rather than between an assembly routine and a Pascal host program. Just as with .PRIVATE and .PUBLIC , these external references must eventually be resolved by the Linker. If such resolution cannot be accomplished, the Linker will indicate the offending label. Naturally, the Assembler cannot be expected to flag these errors, since it has no knowledge of other assemblies.

The host assembly routine must be linked to its external assembly subroutines BEFORE that host assembly routine can be linked into a Pascal host program or UNIT as an EXTERNAL procedure or function.

.DEF Identifies a label that is defined in the current routine as being available for use (by means of .REF) from .PROCs or .FUNCs in other assembly-language routines.

Note: The .PROC and the .FUNC directives also generate a .DEF with the same name. This allows a host assembly routine to call external .PROCs and .FUNCs if the host assembly routine has defined them in a .REF .

FORM: `.DEF <identifierlist>`

EXAMPLE: The following sketched-out routine declares a .DEF for the labels DOIT and THINK . The subroutines bearing the labels DOIT and THINK may then be used by other assembly routines (see example for .REF).

```
        .PROC FARKLE,3
        .DEF DOIT,THINK
             .
             .
            BNE THINK
             .
DOIT        LDA
             .
            RTS
             .
THINK       LDY
             .
            RTS
             .
            .END
```

.REF Identifies a label used in the current routine which refers to a label declared as available (by means of .DEF) in another routine's .PROC or .FUNC . During the linking process, corresponding .DEFs and .REFs are matched.

Note: The .PROC and the .FUNC directives also generate a .DEF with the same name. This allows a host assembly routine to call external .PROCs and .FUNCs if the host assembly routine has defined them in a .REF .

FORM: .REF <identifierlist>

EXAMPLE: The following sketched-out assembly-language routine declares a .REF for the external label DOIT (DOIT was declared available for such reference by the .DEF in the previous example). It then uses that label just as if it referred to a labelled subroutine within the routine itself.

```
.PROC SAMPLE
.REF DOIT
  .
  .
JSR DOIT
  .
  .
.END
```

Note: The assembly routine containing .PROC FARKLE must be linked from its library codefile into the host assembly routine containing .PROC SAMPLE before SAMPLE can be linked in as an EXTERNAL procedure to a Pascal UNIT or program.

LISTING CONTROL DIRECTIVES

The listing control directives determine what is sent to the output file that is specified at assembly time, in response to the prompt

```
OUTPUT FILE FOR ASSEMBLED LISTING: (<CR> FOR NONE)
```

If no listing output file is specified (by just pressing the RETURN key), then all listing control directives are simply ignored as irrelevent.

.LIST and .NOLIST

Allows selective listing of assembly routines. Listing goes to the specified output file when a .LIST is encountered. The .NOLIST is used to turn off the .LIST option. Listing may be turned on and off repeatedly within an assembly. .LIST is the default state.

FORM: .LIST or .NOLIST

.MACROLIST and .NOMACROLIST

Allows selective listing of macro expansions. In general an assembled listing will contain the textual expansion of a macro if the .MACROLIST option was in effect when the macro was defined. On the other hand, an assembled listing will not contain the textual expansion of a macro if the .NOMACROLIST option was in effect when the macro was defined. These options may be used repeatedly throughout an assembly, to list the expansions of certain macros selectively.

Macro expansion text is flagged in the listing by a # to the left of each expanded line. Comments occurring in the macro definition are not repeated in the expansion. The assembled listing of the EXAMPLE earlier in this chapter shows the macro POP defined on PAGE-0 , and listings of the macro expansion appear on PAGE-1 and PAGE-4 .

When assembling nested macro invocations, listing of textual expansion continues until the Assembler encounters the first macro defined with .NOMACROLIST in effect. Listing does not resume until that macro's invocation is complete, regardless of the listing state of the macros invoked by the non-listing macro.

The .LIST and .NOLIST options take precedence over the .MACROLIST and .NOMACROLIST options. The Assembler defaults to the .MACROLIST state.

FORM: .MACROLIST or .NOMACROLIST

EXAMPLE: .NOMACROLIST

.PATCHLIST and .NOPATCHLIST

Allow control over listing of back-patches made to the code file. These options may be used repeatedly throughout an assembly.

When an undefined label is encountered, the assembled listing shows one * for each hexadecimal digit to be filled in later. For example:

```
0019| 10**            BPL DONE
```

When the forward reference is resolved, the back-patch is listed in the form

```
   0019* 00
   001F| A9 00     DONE  LDA #0
```

where the number to the left of the asterisk is the address of the patched location and the number to the right of the asterisk is that location's new value. See PAGE-1 of the assembled listing of the EXAMPLE, earlier in this chapter, for an illustration of back-patch listing.

.PATCHLIST is the default state.

FORM: .PATCHLIST or .NOPATCHLIST

EXAMPLE: .NOPATCHLIST

.PAGE Allows the programmer to explicitly ask for a top of form page break in the listing.

FORM: .PAGE

EXAMPLE: .PAGE

.TITLE Allows the titling of each page if desired. At the start of each procedure the title is set to blanks and must be reset if title is desired. The title is only cleared at the start of the file. In the EXAMPLE assembly listing earlier in this chapter, the title SYMBOLTABLE DUMP was not set by a .TITLE directive. That heading is always used on pages containing symboltable dumps. Upon assembling a further procedure the heading printed returns to what it was before the symboltable dump.

FORM: .TITLE "<title>"

where <title> is any string of printable ASCII characters, including a space. The length of the string must be less than 80 characters. The double quotes are used as delimiters for the string, so a title may not include the double quote character.

EXAMPLE: .TITLE "QRC12 INTERPRETER"

FILE DIRECTIVE

.INCLUDE   Causes the indicated source file to be included at that point.

FORM:

```
.INCLUDE   <filename>
```

where the filename specifies an assembly-language textfile to be included.

If you don't add the suffix .TEXT the system will add it for you. The last character of the filename must be the last non-space character on that line (no comment may follow on the same line).

CORRECT EXAMPLE:

```
.INCLUDE  SHORTSTART.TEXT
```

CORRECT EXAMPLE:

```
.INCLUDE  SHORTSTART.TEXT
          ; CALLS STARTER
```

INCORRECT EXAMPLE:

```
.INCLUDE  SHORTSTART.TEXT  ; CALLS STARTER
```

The Include-file's text is treated by the assembler just as if you had typed that text into the original file at the position of the .INCLUDE directive. For example, if the included file contains a .END , the assembly is terminated at that point.

Note: For a list of Assembler error messages, see the appendix at the end of this manual.

ASSEMBLER DIRECTIVE SUMMARY

METASYMBOL NOTATION

Square brackets [like this] surround optional elements which you may supply. Angle brackets <like this> surround required elements which you must supply. The metasymbol brackets and the brief definition at the end of each line are not to be typed.

ROUTINE DELIMITING DIRECTIVES

.PROC	<identifier>[,expression]	Begins a procedure.
.FUNC	<identifier>[,expression]	Begins a function.
.END		Ends entire assembly.

LABEL DEFINITIONS AND SPACE-ALLOCATION DIRECTIVES

[label]	.ASCII	"<character string>"	Inserts ASCII of chars.
[label]	.BYTE	[valuelist]	Inserts byte of value.
[label]	.BLOCK	<length>[,value]	Inserts block of value.
[label]	.WORD	<valuelist>	Inserts word of value.
<label>	.EQU	<value>	Assigns value to label.
	.ORG	<value>	Next byte at start of assembly file + value.
	.ABSOLUTE		Precedes 1st .PROC; all .ORGs put next byte at abs. location = value.
	.INTERP		1st loc. of interpreter, in relative-location expressions.

MACRO FACILITY DIRECTIVES

.MACRO	<identifier>	Begins a macro definition.
.ENDM		Ends a macro definition.

CONDITIONAL ASSEMBLY DIRECTIVES

[label]	.IF	<expression>	Begins condit'l assembly. If true, assembles next text [up to .ELSE]; if false, only text after a .ELSE .
	[.ELSE]		
	.ENDC		Ends condit'l assembly.

HOST-COMMUNICATION DIRECTIVES

.CONST	<identifierlist>	Takes value from global const in Pascal host.
.PUBLIC	<identifierlist>	Uses a global variable from the Pascal host.
.PRIVATE	<identifier[:integer] list>	Variable not accessible to the Pascal host. Default :1 word/ident.

EXTERNAL COMMUNICATION DIRECTIVES

.DEF	<identifierlist>	Makes label available to other routines.
.REF	<identifierlist>	Label refers to another routine's .DEF'd label.

LISTING CONTROL DIRECTIVES

.LIST and .NOLIST	Turns assembly listing on and off.
.MACROLIST and .NOMACROLIST	Turns listing of macro expansions on and off.
.PATCHLIST and .NOPATCHLIST	Turns listing of back-patches on and off.
.PAGE	Puts page-feed in listing.
.TITLE "<title>"	Titles each page of current .PROC or .FUNC .

FILE DIRECTIVE

.INCLUDE <filename>	Includes named text file in the assembly.

Note: Additional information can be found in this manual's chapters THE LINKER (Linker information), UTILITY PROGRAMS (installing routines in SYSTEM.LIBRARY), and in the TABLES appendix (Assembler error messages).

CHAPTER 7

THE LINKER

176 INTRODUCTION
176 Diskfiles Needed
178 USING THE LINKER

INTRODUCTION

The Linker is invoked automatically if needed when you type R for R(un, or is invoked explicitly by typing L for L(ink, when at the outermost Command level. The Apple Pascal Linker lets you combine code files, which may be compiled P-code or assembled machine code, into the system workfile or another specified codefile. This provides a way to incorporate certain useful routines into your programs without having to rewrite or even recompile or re-assemble these routines. For example, you might wish to use a fast assembly-language routine for some "real-time" application. This routine could be assembled separately, stored in a library, and eventually accessed via the Linker.

To link in routines (either procedures or functions), a Pascal calling program declares those routines to be EXTERNAL . This notifies the Compiler that the routines may be called, but are not provided yet. The Compiler will inform the system that linking is required before execution. The EXAMPLE in this manual's chapter THE 6502 ASSEMBLER shows an assembly-language procedure and function, a Pascal calling program, and the linking process needed to combine the two portions. For more details about the Linker information stored with codefiles, see this manual's appendix, FILE FORMATS.

The Linker is also used to link in certain kinds of Pascal UNITs. A UNIT is a group of related routines which will be used together to perform a common task. Any Pascal files which reference UNITs or EXTERNAL routines, and which have not yet been linked, may be compiled and saved but will need to be linked before they can be executed.

The UNITs that are provided with the Apple Pascal language, such as TURTLEGRAPHICS and APPLESTUFF , are special INTRINSIC UNITs, which are "prelinked" and are USEd directly from SYSTEM.LIBRARY without linking.

For more information on Pascal UNITs, see the Apple Pascal Language Reference Manual. For information on linking from one assembly routine to another, see this manual's chapter THE 6502 ASSEMBLER.

DISKFILES NEEDED

The following files allow you to use the Linker explicitly:

SYSTEM.LINKER	(any diskette, any drive; required)
Host codefile needing external routines	(any diskette, any drive; default is boot diskette's code workfile SYSTEM.WRK.CODE, any drive)
Library codefiles holding external routines	(any diskettes, any drives; default is boot diskette's library file SYSTEM.LIBRARY, any drive)

The following diskfiles allow you to invoke the Linker automatically, using the R(un command:

Host program needing external routines	(any diskette, any drive; default is boot diskette's workfile SYSTEM.WRK.CODE or .TEXT, any drive)
SYSTEM.COMPILER	(any diskette, any drive; required if host program is a textfile)
SYSTEM.EDITOR	(any diskette, any drive; optional; to fix errors found by Compiler)
SYSTEM.SYNTAX	(boot diskette, any drive; optional messages given on entering Editor)
SYSTEM.LINKER	(any diskette, any drive; required)
SYSTEM.LIBRARY	(boot diskette, any drive; required to contain the needed routines)
SYSTEM.PASCAL	(boot diskette, boot drive; required between Compiling, Linking and eXecuting steps)
SYSTEM.LIBRARY	(boot diskette, boot drive; required if program uses long integers, does file I/O using real numbers or SEEK, or USES Intrinsic Units)
SYSTEM.CHARSET	(any diskette, any drive; required only if program uses WSTRING or WCHAR from TURTLEGRAPHICS)

Any time the Linker is invoked, SYSTEM.LINKER must be available on any diskette in any disk drive. This file is normally found on diskette APPLE2: . When the LINKER prompt line appears, SYSTEM.LINKER is no longer necessary, and the diskette containing SYSTEM.LINKER may be removed from the system to make room for other diskettes.

If you attempt to R(un a text workfile, first the Compiler is invoked, which requires that the file SYSTEM.COMPILER be available in any diskette in any disk drive. SYSTEM.COMPILER is normally found on APPLE2: and also on APPLE0: . Then, following successful compilation, the Linker is called (if linking is needed), using SYSTEM.LINKER . The Linker automatically tries to find any needed UNITs or EXTERNAL routines by looking in the file SYSTEM.LIBRARY , which must be on the boot diskette (APPLE1: or APPLE0:) but may be in any disk drive. Finally, following successful compilation and linking the program is executed. If SYSTEM.LIBRARY is required for execution, it must be in the boot drive on the boot diskette.

Note: The system returns to the Command level for an instant between any two portions of the R(un sequence. Therefore, you must normally leave the boot diskette in the boot drive during the entire sequence.

If the workfile has already been compiled into its code version, R(un will not call the Compiler, and SYSTEM.COMPILER is not needed. If you invoke the Linker by typing L , you can link routines that are found in any available disk file. In that case, the file SYSTEM.LIBRARY may not be needed.

Multi-drive note: On multiple-drive systems, diskette APPLE1: is normally your boot diskette. If APPLE1: is in the boot drive, and APPLE2: is in a non-boot drive, your system will have available all the diskfiles it needs to E(dit, C(ompile or A(ssemble, L(ink, X(ecute and R(un.

Two-drive note: To L(ink when the host and library files are not already on APPLE1: or APPLE2: , you can use the Filer to T(ransfer the needed files onto APPLE2: before linking. Alternatively, if the COMMAND prompt line is showing, if L(inking is your only task, and if all your host and library files are on another diskette such as MYDISK: , you could put MYDISK: in the boot drive and APPLE2: in the non-boot drive. When the linking process is complete, the system will return to the Command level. Since your boot diskette is not the boot drive, you will be prompted to put it in.

One-drive note: To R(un a text workfile that needs linking to an external routine, you will have to use the Filer to T(ransfer SYSTEM.LINKER from APPLE2: onto your boot diskette APPLE0: . With this version of APPLE0: in the disk drive, your system will have available all the diskfiles it needs to E(dit, C(ompile or A(ssemble, L(ink, X(ecute and Run. Unfortunately, this will leave only 17 blocks free on APPLE0: for your text and code workfiles, etc. To make more room on your boot diskette, you may wish to remove the files SYSTEM.SYNTAX (use the compiler error messages shown in the Apple Pascal Language Reference Manual, instead), SYSTEM.CHARSET (only needed if your program uses WCHAR or WSTRING from TURTLEGRAPHICS), and even SYSTEM.FILER (can be read in from any diskette, as long as that diskette is in the drive when you invoke the Filer).

USING THE LINKER

There are two different ways to invoke the Linker: by typing L for L(ink or by typing R for R(un, both from the outermost Command level of Pascal.

If the Pascal program in the current text workfile contains EXTERNAL declarations, or USES UNITs which are not INTRINSIC UNITs, typing R for R(un from the outermost Command level automatically invokes the Linker after the Compiler. The Linker automatically uses as its host file the code file where the Compiler saved the code that resulted from a successful compilation (even if that file is not the code workfile SYSTEM.WRK.CODE). When invoked by the R(un command, the Linker automatically searches the file SYSTEM.LIBRARY, which must be on the boot diskette, for the routines or UNITs specified, and links them into the workfile. If the UNIT or EXTERNALly declared routine is not present in SYSTEM.LIBRARY, the Linker will respond with an appropriate message:

```
        UNIT,
        PROC,
        FUNC,
        GLOBAL,
     or PUBLIC <identifier> UNDEFINED
        TYPE <SP>(CONTINUE), <ESC>(TERMINATE)
```

You can press the spacebar, and the Linker will proceed, trying to link whatever routines or UNITs are available in SYSTEM.LIBRARY . Later, you can use the Linker explicitly to link in the remaining routines or UNITs. If the file SYSTEM.LIBRARY is not available on the boot diskette, this message appears:

```
        NO FILE *SYSTEM.LIBRARY
        TYPE <SP>(CONTINUE), <ESC>(TERMINATE)
```

If the Linker fails to find a file with the exact filename specified at any point and that filename does not end in .CODE or in .LIBRARY, it then adds the suffix .CODE to the filename, and tries again. In this case, its own internal specification told it to look for *SYSTEM.LIBRARY . After this message, the Linker does not allow you to specify a different library file for your routine or UNIT, so there is little point in continuing. Just press the ESC key to go back to Command level.

The Linker may also be invoked explicitly, and, in fact, must be invoked explicitly in cases where

(1) the host file into which UNITs or EXTERNAL routines are to be linked is not the code file resulting from a successful compilation initiated by the R(un command, or

(2) the UNITs or EXTERNAL routines to be linked reside in files other than the boot diskette's SYSTEM.LIBRARY .

In order to invoke the Linker explicitly, you type L for L(ink at the Command level and receive the prompt:

```
        LINKING...



        LINKER II.1 [A4]
        HOST FILE?
```

The hostfile is usually the Pascal program codefile into which the external routines or UNITs are to be linked. (In linking between two assembly routines, the hostfile is the routine which used .REF to declare certain labels as external.)

If you just press the RETURN key in response to the prompt, the Linker uses the boot diskette's workfile SYSTEM.WRK.CODE as the hostfile. If the R(un command has just caused the Compiler to save a compiled codefile, that file is taken as the hostfile even if it is not SYSTEM.WRK.CODE . You may also respond by typing the file specification of any other host codefile. If the Linker cannot find a file with the exact filename you typed and that filename does not end in .CODE or in .LIBRARY, it adds the suffix .CODE to the filename and tries again. For this reason, if you respond by typing the non-existent filename

```
MYDISK:MYFILE.CODE
```

the Linker returns the message

```
NO FILE MYDISK:MYFILE.CODE
```

The Linker always displays the full name of the last file it tried to open.

The Linker then asks for the name of the first library file in which the needed Pascal UNITs or EXTERNAL routines (or .DEF assembly subroutines) are to be found:

```
LIB FILE?
```

You should respond by typing the file specification of any codefile containing a Pascal UNIT or EXTERNAL routine that you want linked into the Pascal host program. (In linking between two assembly routines, the library file contains the routine which used .DEF to declare certain labels as available for external use by the host routine.)

The Linker looks first for the exact filename that you type, and then (if the search was unsuccessful) adds the suffix .CODE and looks again. In any case, it always displays the name of the file actually opened. When the specified file has been found, you are given the same prompt again, asking for the filename of another file containing a needed UNIT or routine. Up to eight library files may be referenced in one linking operation. Typing * (and then pressing the RETURN key) in response to a request for a libfile name will cause the Linker to reference SYSTEM.LIBRARY on the boot diskette.

EXAMPLE:

```
LIB FILE? *
OPENING SYSTEM.LIBRARY
```

For information on LIBRARIES and the LIBRARIAN see this manual's chapter UTILITY PROGRAMS. If a host codefile or a library codefile is not of an appropriate type, the Apple will display an error message. These files must contain either compiled Pascal P-code or assembled 6502 assembly code.

When all relevant library files have been specified, answer the next LIB FILE? prompt by just pressing the RETURN key to proceed. The Linker will now prompt with:

```
MAP FILE?
```

If you respond by typing a file specification, the Linker writes a "mapfile" to the file that you have just specified. Note that the suffix .TEXT is appended to the specified filename unless it already ends in .TEXT or a period (.) is the last letter of the filename. The mapfile contains relevant Linker information regarding the linking process. Responding to this prompt by simply pressing the RETURN key causes no mapfile to be written. This is the response you will normally use. The mapfile is a diagnostic and system programming tool, and is not required for most uses of the Linker.

Note: The output codefile (see below) is opened with the [0] filesize and the mapfile is opened AFTER opening the output codefile. If your system tries to put both files on the same diskette, it may be unable to open the mapfile since the output codefile may then occupy all the remaining diskette space. This does not stop the linking process, but you will have no mapfile. You can solve this problem by sending the mapfile to another diskette, to CONSOLE: , etc.

The Linker now reads all segments required to start the linking process. Then you are prompted to type a file specification for saving the linked code output:

```
OUTPUT FILE?
```

(this will often be the same filename as that of the host file, but you may not use the $ same-name option offered by the Compiler and Assembler). It is not necessary to add the suffix .CODE ; that suffix is automatically supplied if you don't type it. After the output file specification has been typed, press the RETURN key and linking will commence. Responding with no filename (by pressing the RETURN key only) causes the linked output to be saved in the boot diskette's workfile, SYSTEM.WRK.CODE .

Note: unless you specify a different filesize, the output codefile is opened with the [0] filesize.

During the linking process, the Linker will report on all segments being linked and on all external routines being copied into the output codefile. The linking process will be stopped if any required segments or routines are missing or undefined. You will be told what was missing, by messages as described at the beginning of this section, and allowed to terminate or continue the linking process.

CHAPTER 8

UTILITY PROGRAMS

184 INTRODUCTION
184 FORMATTING NEW DISKETTES
184 Diskfiles Needed
185 Using the Utility
186 THE SYSTEM LIBRARIAN
187 Diskfiles Needed
188 Example: Installing a UNIT or Routine into a Library File
193 Using the New Library
193 Shorthand Filenames
194 LIBRARY MAPPING
194 Diskfiles Needed
195 Using the Utility
197 Example: Map of SYSTEM.LIBRARY
199 SYSTEM RECONFIGURATION
199 Diskfiles Needed
200 Using the Utility
201 External Terminal Requirements
203 Miscellaneous Information
204 General Terminal Information
204 Control Key Information
207 Video Screen Control Characters
208 List of All SETUP Parameters
210 CHANGING GOTOXY COMMUNICATION
211 Diskfiles Needed
211 Example: Setup For SOROC IQ120
214 REMOVING LINEFEED FROM RETURN
214 Diskfiles Needed
215 Using the Utility
215 Easier Use of the Utility
216 CALCULATOR
216 Diskfiles Needed
217 Using the Utility
218 UTILITIES SUMMARY
218 Formatting New Diskettes
218 The System Librarian
219 Library Mapping
219 System Reconfiguration
220 Changing GOTOXY Communication
220 Removing Linefeed from Return
221 Calculator

INTRODUCTION

In the Apple Pascal operating system, the most often used program portions can be selected from the various prompt lines. Other programs, written to accomplish less commonly needed tasks, are available through the X(ecute command, and new features can be added to the operating system at any time in this way. Several of Apple Pascal's additions to the operating system, called Utility Programs, are described in this chapter.

FORMATTING NEW DISKETTES

Before a new diskette (or one used in a system other than Apple Pascal) can be used with the Apple Pascal system, it must first be "formatted". This means that the diskette is erased, timing marks are recorded on the diskette for the system's reference, addresses are stored to identify each sector and block, and zeroes are stored in all data locations. Then, the diskette's bootstrap program is stored in blocks 0 and 1 (on the outermost track). Finally, a diskette directory is written, and the diskette is given the volume name BLANK: .

DISKFILES NEEDED

The following diskfiles allow you to use the diskette formatting utility program:

FORMATTER.CODE	(any diskette, any drive; required only to start)
FORMATTER.TEXT	(any diskette, any drive; required only to start)
Diskette(s) to be Formatted	(any drive; insert each diskette when prompted, remove when prompted for next diskette to be formatted)

The file FORMATTER.CODE is normally found on diskette APPLE3: . When you terminate the formatting utility program, your boot diskette should be in the boot drive. If it is not there, the system will tell you to

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette).

One-drive note: You can start the diskette formatting utility by X(ecuting FORMATTER with APPLE3: in the drive. When the utility's first prompt line appears, you can then remove APPLE3: from the drive and put in the first diskette to be formatted. Do not remove the diskette being formatted until you are again prompted with

"FORMAT WHICH DISK?". Put the boot diskette back in the drive before you quit the utility program.

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive to X(ecute APPLE3:FORMATTER. When the diskette formatting utility's first prompt line appears on the screen, you can then remove APPLE3: from its drive and put in the first diskette to be formatted.

USING THE UTILITY

From the Command level, with diskette APPLE3: in any available drive, type X for X(ecute. When you are prompted

```
EXECUTE WHAT FILE?
```

respond by typing

```
APPLE3:FORMATTER
```

(Note that you do not need to specify FORMATTER.CODE ; the .CODE suffix is added automatically if you don't type it.) The system then executes FORMATTER.CODE, and displays the following message:

```
APPLE DISK FORMATTER PROGRAM

FORMAT WHICH DISK (4, 5, 9..12) ?
```

You may now remove diskette APPLE3: from its drive, if you wish. Place in any available disk drive the new or used diskette that you wish to format. Then type the volume NUMBER of that disk drive. For example, if you put your new diskette in drive #5: , you should respond by typing

```
5
```

and pressing the RETURN key. First, the program checks the diskette to be sure you are not accidentally re-formatting (and thereby erasing) a diskette previously formatted by the Apple Pascal system. If you forget and leave APPLE3: in the specified drive, for example, you will be warned by the question

```
DESTROY DIRECTORY OF APPLE3 ?
```

If you type N for No, you will again be asked "FORMAT WHICH DISK?".

If all goes well, the disk whirrs and clacks, and this message appears:

```
NOW FORMATTING DISKETTE IN DRIVE 5
```

When formatting is complete, you will be prompted to specify the next diskette to be formatted:

```
FORMAT WHICH DISK (4, 5, 9..12) ?
```

Again, put in any drive the next diskette to be formatted, and then type that drive's volume number.

When you have formatted all the diskettes you wish to format, respond to the prompt

```
          FORMAT WHICH DISK (4, 5, 9..12) ?
```

by just pressing the RETURN key to quit the formatting program. Be sure that your boot diskette is in the boot drive before you quit the formatting utility program, or your system may "hang". (If that happens, put your boot diskette in the boot drive and press the RESET key.)

If the program has trouble formatting a diskette, this message is displayed:

```
          ERROR: UNABLE TO FORMAT DISK.
                 DISKETTE WRITE PROTECTED,
                 BAD MEDIA, OR BAD DRIVE.
```

Check the obvious causes, such as no diskette in that drive, or improper insertion of the diskette. Occasionally, this message is given when a used diskette is re-formatted. If you suspect that is the only cause of trouble, just try re-formatting the diskette again until the old information is completely erased.

THE SYSTEM LIBRARIAN

LIBRARY.CODE is a utility program on diskette APPLE3: that allows you to link separately compiled Pascal UNITs and separately compiled or assembled routines into a library file. If your library file is named SYSTEM.LIBRARY and is on the boot diskette, the R(un command will automatically Link needed UNITs and external routines from your library into the program being R(un. UNITs which are labelled INTRINSIC (see the Apple Pascal Language Reference Manual) are found in your library and used without Linking.

If the R(un command does not find a needed item in the boot diskette's SYSTEM.LIBRARY, you can either L(ink the item in manually (see this manual's chapter THE LINKER) or you can put the item into a new boot diskette library file named SYSTEM.LIBRARY . To add a new UNIT or routine to the boot diskette's SYSTEM.LIBRARY (or to delete one, or even just to move one within the library), it is first necessary to create a new, empty library file. Next, you must link each item that you want from the original SYSTEM.LIBRARY into the new library file. You may then add new items by linking from other codefiles into the new library file being created. In general, your new library file is not created with the filename SYSTEM.LIBRARY . Before the system can use your new library file automatically, you must store your library on the boot diskette and name it SYSTEM.LIBRARY .

DISKFILES NEEDED

The following diskfiles allow you to use the librarian utility:

LIBRARY.CODE	(any diskette, any drive; required only to start)
Link Codefile(s) containing UNITs and routines to put in new library	(any diskettes, any drives; * specifies boot diskette's SYSTEM.LIBRARY, any drive; each file must be available until next prompt for LINK CODE FILE)
Output Codefile for storing new library	(any diskette, any drive; * specifies boot diskette's SYSTEM.LIBRARY, any drive; must be available throughout)

The file LIBRARY.CODE is normally found on diskette APPLE3: . When you Q(uit or A(bort the librarian utility program, your boot diskette should be in the boot drive. If it is not there, the system will tell you to

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette).

One-drive note: On single-drive systems, one approach is to use the Filer to T(ransfer all necessary files onto your boot diskette before X(ecuting LIBRARY.CODE . This is safe because the boot diskette (and the necessary file SYSTEM.PASCAL) is available when the system returns to COMMAND level after using the librarian. However, you can start the librarian by X(ecuting APPLE3:LIBRARY with APPLE3: in the drive. When the librarian's first prompt line appears, you can put any other diskette in the drive. For one-drive use, the Output Code file and all Link Code files must be on the same diskette. If that diskette is not the boot diskette, put your boot diskette back into the drive before you Q(uit the librarian utility program.

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive. When the utility's first prompt line appears on the screen, you can then remove APPLE3: from its drive and put in any other diskettes as needed. The diskette containing the Output Code file must remain in its drive throughout use of the librarian utility.

All note: If the Output Code file's diskette is removed from its drive while using the librarian, all displays will indicate that the new library is still being put together correctly. However, the indicated transfers of items from the input Link Code files to the Output Code file are not actually carried out, even if the Output Code file's diskette is placed in a drive just before Q(uitting the program.

EXAMPLE: INSTALLING A UNIT OR ROUTINE INTO A LIBRARY

Suppose you wish to create a new library file, called NEW.LIBRARY , on diskette MYDISK: . You want this new library file to contain all the items currently in the boot diskette's APPLE1:SYSTEM.LIBRARY , and you wish to add a regular UNIT or assembly-language routine called PILFER from the file MYDISK:UPILFER.CODE.

One-drive note: If yours is a one-drive system, you must first use the Filer to T(ransfer the file UPILFER.CODE from MYDISK: to your boot diskette APPLE1: . Then Q(uit the Filer. From here on, substitute APPLE1: wherever the example says MYDISK: .

From the Command level, with APPLE3: in any available disk drive, type X for X(ecute. When you are prompted

```
EXECUTE WHAT FILE?
```

respond by typing

```
APPLE3:LIBRARY
```

(Note that you do not need to specify LIBRARY.CODE; the .CODE suffix is supplied automatically.) The system then executes LIBRARY.CODE, and soon displays the message

```
PASCAL SYSTEM LIBRARIAN II.1 [B1]
```

At this point, you may remove APPLE3: from its drive. Next the program prompts you for the name of an

```
OUTPUT CODE FILE ->
```

which will become your new library file. For this example, with MYDISK: in any available drive, respond with the name

```
MYDISK:NEW.LIBRARY
```

One-drive note: you should respond APPLE1:NEW.LIBRARY .

This filename is used exactly as you type it; no suffix is added by the system. The diskette containing the Output Code file must remain in its drive throughout use of the librarian.

The program now asks for the name of a

```
LINK CODE FILE ->
```

which will provide the first source of UNITs and routines to link into NEW.LIBRARY . The correct response here, with APPLE1: in any drive, is

```
APPLE1:SYSTEM.LIBRARY
```

(You can also type * to specify the boot diskette's SYSTEM.LIBRARY .)

The screen next displays the "slot number", segment number (in parentheses), name, and length in bytes of each UNIT or routine currently in the input Link Code File (right now, SYSTEM.LIBRARY). There are a maximum of 16 "slots", each containing one code or data segment, in any Apple Pascal program or library file. Note that an Intrinsic UNIT may occupy two slots, one for the code segment and one for the data segment. The number of bytes given for an item is its length in the library. This length includes the number of bytes the item will occupy when linked into your program, plus a considerable number of bytes of Linker information that is not placed in your program.

The screen now looks something like this (you may, of course, have to use CTRL-A to see the right half of the display):

```
SLOT TO LINK AND <SPACE>, = FOR ALL, ? FOR SELECT, N(EW FILE, Q(UIT,
A(BORT

LINK CODE FILE -> APPLE1:SYSTEM.LIBRARY
           0-(30) LONGINTI  2452          8-                  0
           1-(31) PASCALIO  1238          9-                  0
           2-(29) TRANSCEN  1168         10-                  0
           3-(22) APPLESTU   662         11-                  0
           4-(20) TURTLEGR  5202         12-                  0
           5-(21) TURTLEGR   386         13-                  0
           6-                  0         14-                  0
           7-                  0         15-                  0

OUTPUT CODE FILE -> MYDISK:NEW.LIBRARY
CODE FILE LENGTH - 1
```

You now type the slot number, a number from 0 through 15 taken from the leftmost column of the Link Code File display, of an item that you wish placed in the new library file. Then press the spacebar to terminate your entry. Next, when you are prompted

```
SLOT TO LINK INTO?
```

type the number of the slot which the previously specified item should occupy when it is placed in the Output Code File (i.e. NEW.LIBRARY). And again, press the spacebar to terminate your entry. At this time, the transfer of the specified item is carried out.

NOTE: you may abandon your attempt to make a new library at any time, simply by typing A for A(bort instead of a slot number.

For each item placed in the new library file, the Librarian reads that item from the specified slot of the input Link Code File and stores it in the specified slot of the Output Code File. Items may be placed in any available library slot, in any order. After each item is transferred, the librarian changes the display for the current state of the Output Code File, which is your new library file. If you attempt to put two input items into one output slot, this message appears:

```
WARNING - SLOT xx ALREADY LINKED.  PLEASE RECONFIRM (Y/N) -
```

If you type Y , the item previously placed in the specified slot will be replaced by the item currently being moved. Type N to abandon this move.

Note that the old item is NOT removed from the library file you are making even though it no longer appears in any display of the file's contents. This extra code, which makes your new library file larger than it needs to be, will disappear when you make another new library from this one.

To copy every item from the old SYSTEM.LIBRARY into your new library file, you may follow this sequence (press the spacebar to terminate each entry):

```
0
SLOT TO LINK INTO?  0
1
SLOT TO LINK INTO?  1
2
SLOT TO LINK INTO?  2
3
SLOT TO LINK INTO?  3
4
SLOT TO LINK INTO?  4
5
SLOT TO LINK INTO?  5
```

Or you may use one of the other options given in the prompt line. Type an equals sign (=) to quickly copy every item from its slot in the input Link Code File into the same slot in the Output Code File. If you type a question mark (?) the librarian will ask you about each input item in turn:

```
        COPY SLOT 0?
```

Type Y if you wish the item in slot 0 to be copied into slot 0 of your new library file, or type N if you do not wish to copy that item. When you are using the = or the ? to copy items, each item copied from a slot in the input library will automatically be placed in the slot of the same number in your new output library.

When all of the items that you want from this input Link Code File have been copied into the Output Code File, a new input file is requested by typing N for N(ew file. You are again prompted for a

```
            LINK CODE FILE ->
```

In this example, a separately compiled regular Pascal UNIT called PILFER is assumed to exist in a codefile called UPILFER.CODE , but the process would be identical if PILFER were a separately assembled machine-language procedure.

Type the name of this new input Link Code File:

```
            MYDISK:UPILFER
```

One-drive note: You should type APPLE1:UPILFER

The librarian first looks on the specified diskette for a file whose filename is exactly as you typed it. If there is no file with that exact filename and that filename does not end in .CODE , the suffix .CODE is added to the filename and the search is repeated. If the search is still unsuccessful, one of the following messages is displayed:

```
            I/O ERROR # 10          (your file was not found)
            I/O ERROR # 9           (your diskette was not found)
```

In either case, you are prompted to try again. The only way to escape the program at this point is by typing a correct file specification or by pressing the RETURN key and then typing A for A(bort.

When you correctly type the name of this new Link Code File, the following display appears:

```
LINK CODE FILE -> MYDISK:UPILFER.CODE
           0-                  0          8-                 0
           1-                  0          9-                 0
           2-                  0         10-(25) PILFER    362
           3-                  0         11-                 0
           4-                  0         12-                 0
           5-                  0         13-                 0
           6-                  0         14-                 0
           7-                  0         15-                 0
```

In this example, the Unit PILFER occurs in UPILFER.CODE's slot number 10 and is to be linked into slot number 7 (any unused slot is equally good) within NEW.LIBRARY . To accomplish this, you should respond (pressing the spacebar after each response):

```
          10
          SLOT TO LINK INTO?  7
```

The final display of the output library segment table is thus:

```
OUTPUT CODE FILE -> MYDISK:NEW.LIBRARY
CODE FILE LENGTH - 39
           0-(30) LONGINTI  2452          8-                 0
           1-(31) PASCALIO  1238          9-                 0
           2-(29) TRANSCEN  1168         10-                 0
           3-(22) APPLESTU   662         11-                 0
           4-(20) TURTLEGR  5202         12-                 0
           5-(21) TURTLEGR   386         13-                 0
           6-                  0         14-                 0
           7-(25) PILFER     362         15-                 0
```

The new library's length in blocks is displayed and in this example is 39.

Once the needed items from all input Link Code Files have been put into your new library's Output Code File, you lock the new library by typing Q for Q(uit. This question appears at the bottom of the screen:

```
          NOTICE?
```

This gives you the chance to place a copyright notice in your library file. The notice will be displayed when a library map is produced for your file (see the next section of this chapter), and provides an identifying line. You might type

```
          COPYRIGHT 1979 APPLE COMPUTER AND ME
```

for example (any message, up to the end of the current typing line). If you do not want a copyright in your library file, simply press the RETURN key. When the COMMAND prompt line re-appears, your new library is complete.

USING THE NEW LIBRARY

Before you can use your new library, the old SYSTEM.LIBRARY on your boot diskette should be either removed or renamed. Then your new library file must be T(ransferred onto your boot diskette, and its name C(hanged from NEW.LIBRARY to SYSTEM.LIBRARY . You should then I(nitialize the system so that the system will "learn" about the new library's contents and diskette position.

SHORTHAND FILENAMES

In response to the initial prompt "OUTPUT CODE FILE ->" we could have just as easily said SYSTEM.LIBRARY followed by another SYSTEM.LIBRARY in response to the prompt "LINK CODE FILE ->". If you do this, however, the original SYSTEM.LIBRARY will be removed automatically upon completion of the linking process. Typing just * in response to "OUTPUT CODE FILE ->" and again in response to "LINK CODE FILE ->" is an abbreviated way to indicate that the old SYSTEM.LIBRARY will simply be replaced by the new SYSTEM.LIBRARY .

The system only "learns" about the new material in SYSTEM.LIBRARY when the system is booted. If you specified SYSTEM.LIBRARY or * as the Output Code File, the message

```
          PLEASE RE-INITIALIZE SYSTEM
```

appears after you respond to the NOTICE? prompt. Just press the Apple's RESET key.

LIBRARY MAPPING

The library mapping utility program produces a map of a library file (or any codefile) and lists the Linker information maintained for each segment of the file. In the case of segments which are Pascal UNITs the map file will also contain the interface section of the UNIT. See this manual's chapter THE LINKER for more information. See the Apple Pascal Language Reference Manual for greater detail about UNITs and linkage to external routines.

DISKFILES NEEDED

The following diskfiles allow you to use the library mapping utility program:

LIBMAP.CODE	(any diskette, any drive; required only to start)
Library Codefile(s) to be mapped	(any diskettes, any drives; * specifies boot diskette's SYSTEM.LIBRARY, any drive; each file must be available until next prompt for LIBRARY NAME)
Map output textfile	(any diskette, any drive, or any other output device; default is CONSOLE: ; must be available throughout)

The file LIBMAP.CODE is normally found on diskette APPLE3: . When you terminate the library mapping utility program, your boot diskette should be in the boot drive. If it is not there, the system will tell you to

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette).

One-drive note: On single-drive systems, one approach is to use the Filer to T(ransfer all necessary files onto your boot diskette before X(ecuting LIBMAP.CODE . This works well because the boot diskette (and the necessary file SYSTEM.PASCAL) is available when you return to COMMAND level after using the utility. However, you can X(ecute LIBMAP.CODE with APPLE3: in the drive. When the first prompt line appears, you can put any other diskette in the drive. If you are storing the Map Output textfile on diskette, you must place all the input Library codefiles on the same diskette as the Map Output textfile. If you specify the Map Output File Name as PRINTER: or CONSOLE: you may put in the drive, one at a time, the diskettes containing the input Library codefiles, leaving each diskette in the drive while Library files on that diskette are being mapped. Put the boot diskette back in the drive before you quit the library mapping program.

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive. When the utility's first prompt line appears on the screen, you can then remove APPLE3: from its drive and put in any other diskettes as needed. If you are storing the Map Output textfile on diskette, that diskette must remain in its drive throughout the mapping procedure.

USING THE UTILITY

With the COMMAND prompt line showing, and with diskette APPLE3: in any available disk drive, type X for X(ecute. When the prompt

```
EXECUTE WHAT FILE?
```

appears, respond by typing

```
APPLE3:LIBMAP
```

(Note that you do not need to specify LIBMAP.CODE ; the suffix .CODE is supplied automatically if you don't type it). Soon this message appears

```
LIBRARY MAP UTILITY  II.1 [A2]
```

and the program prompts you to

```
ENTER LIBRARY NAME:
```

When you respond by typing an input Library (or any codefile) file specification, the program first attempts to find the file exactly as specified. If this search fails, the suffix .CODE is added and the search is made again. If the specified file or diskette is not found, this message appears:

```
BAD FILE
ENTER LIBRARY NAME:
```

Other errors give the message

```
NOT A CODE FILE
ENTER LIBRARY NAME:
```

If you respond by simply typing an asterisk (*), this specifies the file SYSTEM.LIBRARY, on the boot diskette in any drive, as the input Library file .

The library mapping utility is usually used to list library definitions; but the option is also available to include Linker information such as intra-library symbol references. Should this feature be desired, type a Y when queried

```
LIST LINKER INFO TABLE (Y/N)?
```

If you respond by typing a Y , you will also be asked

```
LIST REFERENCED ITEMS (Y/N)?
```

A space (or pressing the RETURN key) is considered an N .

You are now prompted to specify a

```
MAP OUTPUT FILE NAME:
```

Note that if you don't add the suffix .TEXT to the filename, the system automatically will add it for you. To suspend this suffix-adding feature, you must type an extra period after the filename. Responding by pressing only the RETURN key sends the map output to CONSOLE: , by default.

Several libraries may be mapped in succession. These maps will all be sent to the same Map Output File specified for the first input Library file.

To quit the library mapping utility, press the RETURN key the next time you are prompted to

```
ENTER LIBRARY NAME:
```

Be sure your boot diskette is in the boot drive before you quit this utility.

EXAMPLE: MAP OF SYSTEM.LIBRARY

```
    LIBRARY MAP FOR APPLE0:SYSTEM.LIBRARY

       COPYRIGHT 1979 APPLE COMPUTER INC.


Segment #30:
System version = II.1, code type is 6502
LONGINTI  library unit (LINKED INTRINSIC)

  TYPE DECMAX = INTEGER[36];
       STUNT = RECORD CASE INTEGER OF
                 2:(W2:INTEGER[4]);
                 3:(W3:INTEGER[8]);
                 4:(W4:INTEGER[12]);
                 5:(W5:INTEGER[16]);
                 6:(W6:INTEGER[20]);
                 7:(W7:INTEGER[24]);
                 8:(W8:INTEGER[28]);
                 9:(W9:INTEGER[32]);
                10:(W10:INTEGER[36])
               END;

  PROCEDURE FREADDEC(VAR F: FIB; VAR D: STUNT; L: INTEGER);
  PROCEDURE FWRITEDEC(VAR F: FIB; D: DECMAX; RLENG: INTEGER);

     --------------------------------------------------

Segment #31:
System version = II.1, code type is Undefined
PASCALIO  library unit (LINKED INTRINSIC)

  PROCEDURE FSEEK(VAR F: FIB; RECNUM: INTEGER);
  PROCEDURE FREADREAL(VAR F: FIB; VAR X: REAL);
  PROCEDURE FWRITEREAL(VAR F: FIB; X: REAL; W, D: INTEGER);

     --------------------------------------------------

Segment #29:
System version = II.1, code type is P-Code (most sig. 1st)
TRANSCEN  library unit (LINKED INTRINSIC)
{$  }

  FUNCTION SIN(X:REAL):REAL;
  FUNCTION COS(X:REAL):REAL;
  FUNCTION EXP(X:REAL):REAL;
  FUNCTION ATAN(X:REAL):REAL;
  FUNCTION LN(X:REAL):REAL;
  FUNCTION LOG(X:REAL):REAL;
  FUNCTION SQRT(X:REAL):REAL;

     --------------------------------------------------
```

```
Segment #22:
System version = II.1, code type is 6502
APPLESTU  library unit (LINKED INTRINSIC)
{$  }

    FUNCTION PADDLE(SELECT: INTEGER): INTEGER;
    FUNCTION BUTTON(SELECT: INTEGER): BOOLEAN;
    PROCEDURE TTLOUT(SELECT: INTEGER; DATA: BOOLEAN);
    FUNCTION KEYPRESS: BOOLEAN;
    FUNCTION RANDOM: INTEGER;
    PROCEDURE RANDOMIZE;
    PROCEDURE NOTE(PITCH,DURATION: INTEGER);

     ------------------------------------------------------------

Segment #20:
System version = II.1, code type is 6502
TURTLEGR  library unit (LINKED INTRINSIC)

    TYPE
      SCREENCOLOR=(none,white,black,reverse,radar,black1,
                   green,violet,white1,black2,orange,blue,white2);

    PROCEDURE INITTURTLE;
    PROCEDURE TURN(ANGLE: INTEGER);
    PROCEDURE TURNTO(ANGLE: INTEGER);
    PROCEDURE MOVE(DIST: INTEGER);
    PROCEDURE MOVETO(X,Y: INTEGER);
    PROCEDURE PENCOLOR(PENMODE: SCREENCOLOR);
    PROCEDURE TEXTMODE;
    PROCEDURE GRAFMODE;
    PROCEDURE FILLSCREEN(FILLCOLOR: SCREENCOLOR);
    PROCEDURE VIEWPORT(LEFT,RIGHT,BOTTOM,TOP: INTEGER);
    FUNCTION  TURTLEX: INTEGER;
    FUNCTION  TURTLEY: INTEGER;
    FUNCTION  TURTLEANG: INTEGER;
    FUNCTION  SCREENBIT(X,Y: INTEGER): BOOLEAN;
    PROCEDURE DRAWBLOCK(VAR SOURCE; ROWSIZE,XSKIP,YSKIP,WIDTH,HEIGHT,
                        XSCREEN,YSCREEN,MODE: INTEGER);
    PROCEDURE WCHAR(CH: CHAR);
    PROCEDURE WSTRING(S: STRING);
    PROCEDURE CHARTYPE(MODE: INTEGER);

     ------------------------------------------------------------

Segment #21:
System version = II.1, code type is P-Code (least sig. 1st)
TURTLEGR  data segment

     ------------------------------------------------------------
```

SYSTEM RECONFIGURATION

The Apple Pascal Operating System keeps certain information about the configuration of your system in a file called SYSTEM.MISCINFO . During each system initialization this file is read into memory, and from there it is used by many parts of the system, particularly by the Editor.

SYSTEM.MISCINFO comes already set up to work correctly with your Apple's keyboard and its TV or monitor display, and you can operate the system without ever having to study this section of the manual.

In addition, the language system diskette APPLE3: contains a file named SOROC.MISCINFO , which contains the configuration information necessary to run the Apple Pascal system with a Soroc IQ120 external terminal, and another file named HAZEL.MISCINFO , which contains the configuration information for a Hazeltine 1500 external terminal. To use either of those terminals, it is only necessary to rename the old SYSTEM.MISCINFO, and then change the name of either SOROC.MISCINFO or HAZEL.MISCINFO (the one corresponding to your terminal) to SYSTEM.MISCINFO . Finally, you must read the next section of this chapter, CHANGING GOTOXY COMMUNICATION, which tells you how to bind a new GOTOXY routine into SYSTEM.PASCAL . That section has a complete example for setting up the Apple to use a SOROC IQ120 terminal.

You only need to read the rest of this section if you are going to use the Apple Pascal system with an external terminal, and that external terminal is neither a Soroc IQ120 nor a Hazeltine 1500.

DISKFILES NEEDED

The following diskfiles allow you to use the system reconfiguration utility program:

SETUP.CODE	(any diskette, any drive; required to start, and also required to be in same drive any time the T(each command is selected)
Output codefile, creates NEW.MISCINFO [1 block]	(boot diskette, any drive; optional)

The file SETUP.CODE is normally found on diskette APPLE3: . All systems will normally start the reconfiguration program by X(ecuting APPLE3:SETUP with APPLE3: in any available disk drive.

IMPORTANT: The T(eaching portion of this utility is a segment procedure overlay, which means the system must re-access SETUP.CODE in its original disk location when you type T for T(each. If you select the T(each command, you must first be sure the diskette containing SETUP.CODE (usually APPLE3:) is still in the drive it occupied when SETUP.CODE was X(ecuted. If it is not there when you type T for T(each, the system may "hang", and may even cause damage to the information on other diskettes in the system. It is not necessary to keep APPLE3: in its drive after you have completed the T(each sequence, or if you do not use the T(each command.

When you E(xit the reconfiguration utility program, your boot diskette should be in the boot drive. If it is not there, the system will tell you

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette).

One-drive note: You will normally put APPLE3: in the disk drive to begin, and leave it there while changing the setup information. When you are ready to Q(uit the reconfiguration utility and do a D(isk update, you can remove APPLE3: from the drive and put in your boot diskette. Your boot diskette must be in the drive if you do a D(isk update, which creates the file NEW.MISCINFO on the boot diskette. Put the boot diskette in the drive before you E(xit the reconfiguration utility program.

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive to begin. Ordinarily, you should leave these disks in their drives throughout the use of the reconfiguration utility.

USING THE UTILITY

If you are going to use an external terminal, certain information needs to be initially set up by you to conform to your particular hardware configuration or to your taste or convenience. Most of this information concerns the nature of your terminal and keyboard, although there are a few miscellaneous fields.

The system reconfiguration utility is run by entering the Command level of the system and, with APPLE3: in any available disk drive, typing X for X(ecute. When the prompt message

```
EXECUTE WHAT FILE?
```

appears, respond by typing the filename

```
APPLE3:SETUP
```

(Note that you do not need to specify SETUP.CODE ; the .CODE suffix is automatically added to any filename you type.) You should then see the following:

```
INITIALIZING...........................
..........................
 SETUP: C(HANGE T(EACH H(ELP Q(UIT [S.2]
```

All commands to the SETUP program are invoked by typing a single letter chosen from the promptline

```
SETUP: C(HANGE T(EACH H(ELP Q(UIT
```

Type H for H(elp in finding out what the commands at this level do.

Type T if you wish the program to T(each you how to use the reconfiguration utility. This command tells you how to enter non-printing characters, how to avoid making a prompted change, how to delete a typing error, how to change the default radix, and other useful information.

IMPORTANT: If you type T for T(each, you must first be sure that APPLE3: is still in the drive it occupied when APPLE3:SETUP.CODE was X(ecuted. If it is not there, the system may "hang", and may even cause damage to the information on other diskettes in the system. It is not necessary to keep APPLE3: in its drive after you have completed the T(each sequence, or if you do not use the T(each command.

Type C if you wish to C(hange or examine the various items of the system's information about your hardware configuration. You may either change a single item that you specify from the LIST OF ALL SETUP PARAMETERS (at the end of this section); or you may choose to have the program step through all the parameters, letting you examine or change each one. The T(each command gives a full explanation of all of these options.

Type Q when you wish to make your configuration changes permanent and leave the reconfiguration program. The reconfiguration utility's Q(uit command offers several options:

D(isk update: creates the file NEW.MISCINFO, on the boot diskette in any drive. This filename must later be changed to SYSTEM.MISCINFO before the new setup can be used by the system. No message is given if the boot diskette is not found, but no file NEW.MISCINFO is created. You are then returned to the Q(uit level of the reconfiguration program.

M(emory update: places the definitions in memory, where they change the system setup until the next boot, RESET, or initialization. You are then returned to the Q(uit level of the reconfiguration program.

R(eturn: takes you back to the main prompt line of the reconfiguration program, in case you are not done.

H(elp: explains the Q(uit options, and then returns you to the Q(uit level of the reconfiguration program.

E(xit: returns you to the operating system's Command level. Put your boot diskette back in the boot drive before you type E .

The operation of the reconfiguration utility is self teaching (just type T for T(each from the main SETUP prompt line), so the rest of this section explains the various items of information that this utility was designed to change.

EXTERNAL TERMINAL REQUIREMENTS

By using an Apple Communications Interface Card and an external terminal such as the Soroc IQ120, it becomes possible to do text and program editing in upper and lower case characters, on a large (80 characters by 24 lines) screen. For maximum effectiveness, the Editor requires a reasonably powerful CRT terminal with the following features:

CLEAR TO END OF LINE

CLEAR TO END OF SCREEN

GOTOXY ADDRESSING - go directly to a given row and column on the screen

- NDFS - non-destructive forward space (the inverse of back-space)

- LF - down one line (and if at the bottom of the screen scrolls up)

- RLF - reverse line feed (up one line; not required to reverse scroll)

The Soroc IQ120, DEC VT52 and Hazeltine 1500 are examples of such terminals. The main advantage of using an external terminal with the Apple Pascal system is that it can provide upper and lower case for text editing, and allows you to see the system's entire eighty-character line at one time. For most programming purposes, an external terminal is completely unnecessary.

The reconfiguration utility does not tell the system how to do random-access cursor addressing on an external terminal (for those terminals which have this capability). To allow the system to use that feature, please refer to the next section, CHANGING GOTOXY COMMUNICATION.

Note: A parameter value of "NUL" (ASCII 0) usually means the parameter does not apply to the system being set up.

MISCELLANEOUS INFORMATION

HAS CLOCK Value: TRUE or FALSE

Will be FALSE for the Apple. No provision has been made for operation with accessory real-time clocks.

STUDENT Value: TRUE or FALSE

If true, tells the system to simplify certain parts of the system for novice use. E.g., an error detected while compiling sends student back to the Editor without choice.

HAS 8510A Value: TRUE or FALSE

This is always FALSE on an Apple.

HAS BYTE FLIPPED MACHINE Value: TRUE or FALSE

Must be FALSE for the Apple.

GENERAL TERMINAL INFORMATION

HAS SLOW TERMINAL Value: TRUE or FALSE.

When this field is true, the system issues abbreviated promptlines and messages. Suggested setting: 600 baud and under -- True, otherwise False. This is normally FALSE on the Apple.

HAS RANDOM CURSOR ADDRESSING Value: TRUE or FALSE

Only applies to video terminals. See Section 4.7 in order to allow the system to make use of this feature. On the Apple, this field is TRUE.

HAS LOWER CASE Value: TRUE or FALSE

This is normally FALSE for an Apple, although it may be true if you have an external terminal with lower case.

SCREEN WIDTH Value: The number of characters per line of a terminal.

For most external terminals, this should be 80. For the Apple with no external terminal, setting a value of 79 allows almost all of the system's 80-character window to be viewed (with the help of CTRL-A), while causing some prompt lines to be displayed in a shortened form that is better suited to a 40-character screen.

SCREEN HEIGHT Value: The number of lines per display screen of a video terminal.

Set to 0 for a hard copy terminal or other terminal in which paging is not appropriate. Some terminals may require you to set the screen to one more than the number of available screen lines to insure proper scrolling. This is set to 24 for the Apple.

NONPRINTING CHARACTER Value: Any printing character.

Specifies what should be displayed by the terminal to indicate the presence of a non-printing character. Recommended setting: ASCII ? .

VERTICAL MOVE DELAY Value: The number of nulls to send after a vertical cursor move.

Many types of terminals require a delay after certain cursor movements which enables the terminal to complete the movement before the next character is sent. This number of nulls will be sent after carriage returns, ERASE TO END OF LINE, ERASE TO END OF SCREEN and MOVE CURSOR UP. This number is 0 on the Apple.

CONTROL KEY INFORMATION

You may choose which control keys suit your particular keyboard arrangement and your taste. Again, this section has already been set up for your Apple.

Some keyboards generate two codes when certain single keys are pressed. If that is the case for any of the keys mentioned here, it must be noted in the field PREFIXED [<fieldname>] which has either the value TRUE or the value FALSE. The prefix for all such keys must be the same and must be noted in the field LEAD IN FROM KEYBOARD. This feature may also be used to access control functions with two-character sequences if your keyboard is unable to generate many control characters. As an example, suppose your keyboard had a vector pad which generated the value pairs ESC-U , ESC-D , ESC-L and ESC-R for the keys for Up-arrow, Down-arrow, Left-arrow and Right-arrow, respectively. Assume also that all other keys on the keyboard generate only single codes. Then the user would give the following fields the following values:

KEY FOR MOVING CURSOR UP	ASCII U
KEY FOR MOVING CURSOR DOWN	ASCII D
KEY FOR MOVING CURSOR LEFT	ASCII L
KEY FOR MOVING CURSOR RIGHT	ASCII R
LEAD IN FROM KEYBOARD	ESC
PREFIXED[KEY FOR MOVING CURSOR UP]	TRUE
PREFIXED[KEY FOR MOVING CURSOR DOWN]	TRUE
PREFIXED[KEY FOR MOVING CURSOR LEFT]	TRUE
PREFIXED[KEY FOR MOVING CURSOR RIGHT]	TRUE

KEY FOR STOP

Console output stop character. The STOP character is a toggle; when pressed, the key will cause output to the file 'OUTPUT' to cease. When the key is depressed again, the write to file 'OUTPUT' will resume where it left off. This function is very useful for reading data which is being displayed faster than one can read. Suggested setting: CTRL-S

KEY FOR FLUSH

Console output cancel character. Similar in concept and usage to the STOP key, the FLUSH key will cause output to the file 'OUTPUT' to go undisplayed until FLUSH is pressed again or the system writes to file 'KEYBOARD'. Note that, unlike the STOP key, processing continues uninterrupted while output goes undisplayed. Suggested setting: CTRL-F

KEY FOR BREAK

Typing the character BREAK will cause the program currently executing to be terminated with a run-time error immediately. Suggested setting: something difficult to hit accidentally. This is set to ASCII 0 on the Apple which, in this case, represents CTRL-@ .

KEY TO END FILE

Console end of file character. When reading from the files KEYBOARD or INPUT or the unit CONSOLE: , this key sets the Boolean function EOF to TRUE. See the description of the EOF intrinsic in the Apple Pascal Language Reference Manual. Suggested setting: ASCII ETX (CTRL-C)

KEY TO DELETE CHARACTER

Each time you press this key one character is removed from the current line, until nothing is left on that line. Suggested setting: ASCII BS (left-arrow key, or CTRL-H)

KEY TO DELETE LINE

Depressing LINE DELETE will cause the current line of input to be erased. Suggested setting: CTRL-X

> The rest of this section contains information only of interest to users who are using video display terminals with a selective erase capability and may be safely ignored by users having any other kind of terminal, such as hardcopy terminals or storage tube terminals.

KEY TO MOVE CURSOR UP
KEY TO MOVE CURSOR DOWN
KEY TO MOVE CURSOR LEFT
KEY TO MOVE CURSOR RIGHT

These keys are used by the screen oriented editor to control the basic motions of the cursor. If the keyboard has a vector pad, set these fields to the values it generates. Otherwise, we suggest that you choose four keyboard keys which lie in the pattern of a vector pad, and use the control codes which correspond to them. For example, the keys ′O′, ′.′, ′K′ and ′;′ on most keyboards encircle an imaginary vector pad. You may wish to use a prefix character before such keys as described above.

On the Apple, of course, the right-arrow and left-arrow keys serve for right and left cursor motion. Because of their vertical orientation, CTRL-O and CTRL-L have been chosen for up and down motion of the cursor.

EDITOR "ESCAPE" KEY

The key which, in the system screen oriented editor, is to be used to escape from commands, reversing any action taken. Suggested setting: ESC

EDITOR "ACCEPT" KEY

The key which, in the system screen oriented editor, is to be used to accept commands, making permanent any action taken. Suggested setting: ASCII ETX (CTRL-C).

VIDEO SCREEN CONTROL CHARACTERS

This section describes the characters which, when sent to the terminal by the computer, control the terminal's actions. You should consult the manual for your terminal to find the appropriate values. If a terminal does not have one of these characters, the field should be set to 0 unless otherwise directed.

Some screens require a two-character sequence to exercise some of their functions. If the first character in all of these sequences is the same, it can be set as the value of the field LEAD IN TO SCREEN and for each <fieldname> which requires that prefix, the user must set the field PREFIX[<fieldname>] to TRUE. For example, suppose ERASE TO END OF LINE and ERASE TO END OF SCREEN were respectively performed by the sequences ESC-L and ESC-S but all the other screen controls were single characters. The user would then set the following fields to the following values:

LEAD IN TO SCREEN	ASCII	ESC
ERASE TO END OF LINE	ASCII	L
ERASE TO END OF SCREEN	ASCII	S
PREFIXED[ERASE TO END OF SCREEN]	TRUE	
PREFIXED[ERASE TO END OF LINE]	TRUE	

ERASE TO END OF SCREEN

The character which erases the screen from the current cursor position to the end of the screen.

ERASE TO END OF LINE

The character which, when sent to the screen, erases all characters from the current cursor position to the end of the line the cursor is on.

ERASE LINE

The character which, when sent to the screen, erases all the characters on the line the cursor is currently on.

ERASE SCREEN

The character which, when sent to the screen, erases the entire screen.

BACKSPACE

The character which, when sent to the screen, causes the cursor to move one space to the left.

MOVE CURSOR HOME

The character which moves your cursor to the upper left of the current page. IMPORTANT: If your terminal does not have such a character, set this field to CARRIAGE RETURN, ASCII mnemonic CR.

MOVE CURSOR UP
MOVE CURSOR RIGHT

The characters which move your cursor non-destructively one space in those directions.

LIST OF ALL SETUP PARAMETERS

Parameter Field Name	Default value for SYSTEM.MISCINFO on APPLE0: or APPLE1:
BACKSPACE	left-arrow key (CTRL-H)
EDITOR ACCEPT KEY	CTRL-C
EDITOR ESCAPE KEY	ESC
ERASE LINE	NUL (ASCII 0)
ERASE SCREEN	CTRL-L
ERASE TO END OF LINE	CTRL-]
ERASE TO END OF SCREEN	CTRL-K
HAS 8510A	FALSE
HAS BYTE FLIPPED MACHINE	FALSE
HAS CLOCK	FALSE
HAS LOWER CASE	FALSE
HAS RANDOM CURSOR ADDRESSING	TRUE
HAS SLOW TERMINAL	FALSE
KEY FOR BREAK	NUL (ASCII 0)
KEY FOR FLUSH	CTRL-F
KEY FOR STOP	CTRL-S
KEY TO DELETE CHARACTER	left-arrow key (CTRL-H)
KEY TO DELETE LINE	CTRL-X

```
KEY TO END FILE                     CTRL-C
KEY TO MOVE CURSOR DOWN             CTRL-L
KEY TO MOVE CURSOR LEFT             left-arrow key  (CTRL-H)
KEY TO MOVE CURSOR RIGHT            right-arrow key (CTRL-U)
KEY TO MOVE CURSOR UP               CTRL-O
LEAD IN FROM KEYBOARD               NUL             (ASCII 0)
LEAD IN TO SCREEN                   NUL             (ASCII 0)
MOVE CURSOR HOME                    CTRL-Y
MOVE CURSOR RIGHT                   CTRL-\
MOVE CURSOR UP                      CTRL-_
NON PRINTING CHARACTER              ?
PREFIXED [DELETE CHARACTER]         FALSE
PREFIXED [EDITOR ACCEPT KEY]        FALSE
PREFIXED [EDITOR ESCAPE KEY]        FALSE
PREFIXED [ERASE LINE]               FALSE
PREFIXED [ERASE SCREEN]             FALSE
PREFIXED [ERASE TO END OF LINE]     FALSE
PREFIXED [ERASE TO END OF SCREEN]   FALSE
PREFIXED [KEY FOR BREAK]            FALSE
PREFIXED [KEY FOR FLUSH]            FALSE
PREFIXED [KEY TO MOVE CURSOR DOWN]  FALSE
PREFIXED [KEY TO MOVE CURSOR LEFT]  FALSE
PREFIXED [KEY TO MOVE CURSOR RIGHT] FALSE
PREFIXED [KEY TO MOVE CURSOR UP]    FALSE
PREFIXED [KEY FOR STOP]             FALSE
PREFIXED [KEY TO DELETE CHARACTER]  FALSE
PREFIXED [KEY TO DELETE LINE]       FALSE
PREFIXED [KEY TO END FILE]          FALSE
PREFIXED [MOVE CURSOR HOME]         FALSE
PREFIXED [MOVE CURSOR RIGHT]        FALSE
PREFIXED [MOVE CURSOR UP]           FALSE
PREFIXED [NON PRINTING CHARACTER]   FALSE
SCREEN HEIGHT                       24
SCREEN WIDTH                        79
STUDENT                             FALSE
VERTICAL MOVE DELAY                 0
```

CHANGING GOTOXY COMMUNICATION

The GOTOXY procedure, which allows the Apple Pascal operating system to communicate with the video screen, is already set up correctly for the Apple. GOTOXY is included in the system as one of the intrinsic procedures in the Apple Pascal language. See the Apple Pascal Language Reference Manual for more details about the intrinsic GOTOXY . This portion of the manual is only presented for your reference, as you will not normally need to change the GOTOXY procedure unless you want to use an external terminal.

If you are going to use an external terminal, you should first read the previous section of this chapter, SYSTEM RECONFIGURATION, then follow the directions given there for creating a new boot diskette file SYSTEM.MISCINFO .

The program BINDER.CODE on diskette APPLE3: alters the file SYSTEM.PASCAL on the boot diskette. You are prompted to provide "GOTOXY", a procedure which must be created and bound into the system (only once) in order to make the system communicate correctly with your external terminal's screen.

On diskette APPLE3: there are examples of Pascal GOTOXY procedures already written for two of the more popular external terminals. The file SOROCGOTO contains the correct GOTOXY procedure for the Soroc IQ120, and the file HAZELGOTO contains a GOTOXY for the Hazeltine 1500. These procedures have already been compiled into their .CODE versions, but the .TEXT versions have been included also, to give you a model which can be modified for use with other terminals.

If the GOTOXY cursor-addressing procedure for your terminal is not already on APPLE3:, you must create one (by modifying SOROCGOTO.TEXT) and compile it. The procedure may NOT be named GOTOXY.

The GOTOXY procedure sends the cursor to a point on the screen determined by a specified pair of coordinates (XCOORD,YCOORD). The procedure assumes the following:

1. A video screen terminal
2. An Apple Pascal system
3. The upper left-hand corner of the screen is X=0, Y=0
4. GOTOXY corrects for bad input data: X-coordinates must be limited to the number of characters per line (integers in the range 0 through 79 for a SOROC IQ120); Y-coordinates must be limited to the number of lines per screen (integers in the range 0 through 23 for a SOROC IQ120).

In writing your own Pascal GOTOXY procedure, here are two common errors:

Possible error:	Possible cure:
Nil memory reference at compile time	Remove the program heading and try again
Value range error when executing BINDER	(*$U-*) should be the first thing in the GOTOXY file

DISKFILES NEEDED

The following diskfiles allow you to use the utility for changing GOTOXY communication with the screen.

BINDER.CODE	(any diskette, any drive; required only to start)
SYSTEM.PASCAL	(boot diskette, any drive; required to start)
Codefile containing new GOTOXY procedure	(any diskette, any drive; required throughout; for SOROC IQ120 use SOROCGOTO.CODE; for Hazeltine 1500 use HAZELGOTO.CODE)
Output codefile, creates NEW.PASCAL [36 blocks]	(boot diskette, any drive; required throughout; can later be used to replace SYSTEM.PASCAL)

The file BINDER.CODE is normally found on diskette APPLE3: . When the utility for changing GOTOXY communication terminates, your boot diskette should be in the boot drive. If it is not there, the system will tell you

```
    PUT IN APPLE1:
```

(if APPLE1: is your boot diskette).

One-drive note: First, use the Filer to T(ransfer APPLE3:BINDER.CODE and the file containing your new GOTOXY procedure (for a SOROC IQ120, this would be APPLE3:SOROCGOTO.CODE) onto your boot diskette. You are then ready to X(ecute BINDER with your boot diskette in the drive.

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive. You are then ready to X(ecute APPLE3:BINDER .

EXAMPLE: SETUP FOR SOROC IQ120

You are about to create a new boot diskette, so you should first make a copy of the current boot diskette APPLE1: .

Now, from the Command level, with all the necessary files in the available disk drives, type X for X(ecute. Answer the question

```
        EXECUTE WHAT FILE?
```

by typing the file name

```
        APPLE3:BINDER
```

One-drive note: You have T(ransferred BINDER.CODE to your boot diskette, APPLE1: . You should type

```
          APPLE1:BINDER
```

(Note that it is not necessary to specify BINDER.CODE ; the .CODE suffix is automatically added if you don't type it). The screen will soon show this title:

```
          APPLE GOTOXY BINDER
```

The program now looks for the old file SYSTEM.PASCAL, which must be on your boot diskette in any drive. If you see this message

```
          ERROR: NO FILE SYSTEM.PASCAL
          PRESS SPACE TO CONTINUE
```

your boot diskette was probably not in any drive. You should put your boot diskette in the boot drive and press the Apple's spacebar to return to Command level. Then you can try to X(ecute the program again.

When the program has successfully found the boot diskette file SYSTEM.PASCAL , it prompts you to specify the

```
          FILE WHICH CONTAINS GOTOXY?
```

For this example, you should respond by typing

```
          APPLE3:SOROCGOTO
```

(for a different terminal, this would be the new GOTOXY procedure you compiled after modifying SOROCGOTO.TEXT for your terminal). The program looks first for a file whose filename is exactly as you typed it. If that search is not successful, the suffix .CODE is added to the filename and the search is made again. When your file is found, the disks whirr, and messages appear saying

```
          COPYING SEGMENT 15
          COPYING SEGMENT 0
          COPYING SEGMENT 1
          COPYING SEGMENT 2
          COPYING SEGMENT 3
          COPYING SEGMENT 4
          COPYING SEGMENT 5
```

and so on. When the COMMAND prompt line reappears, the copy of APPLE1: has the new file NEW.PASCAL on it. This file is the old SYSTEM.PASCAL with the new GOTOXY procedure for your terminal bound into it. Before the system can use this new file, the old file

SYSTEM.PASCAL must be removed from the disk (or at least renamed) and NEW.PASCAL must be given the name SYSTEM.PASCAL . To do this all at once, type F to enter the Filer, and then type T for T(ransfer. The following dialog will then do the job:

```
TRANSFER?  NEW.PASCAL
TO WHERE?  SYSTEM.PASCAL[36]
REMOVE OLD APPLE1:SYSTEM.PASCAL ?  Y
APPLE1:NEW.PASCAL
 --> APPLE1:SYSTEM.PASCAL
```

(Remember that the Apple produces [by typing CTRL-K, and] by typing SHIFT-M .) A copy of NEW.PASCAL has now replaced the old SYSTEM.PASCAL and that copy was renamed at the same time to SYSTEM.PASCAL .

You can now R(emove the original file NEW.PASCAL from the boot diskette. Finally, to avoid confusion, C(hange the name of this new boot diskette from APPLE1: to SOROC1: and label the diskette with this name.

At this time, you should also replace the file SYSTEM.MISCINFO on SOROC1: with the file called APPLE3:SOROC.MISCINFO (for a different terminal, this would be the NEW.MISCINFO you generated with the SETUP program, described in the previous section of this chapter, SYSTEM RECONFIGURATION). This dialog will do it all at once:

```
TRANSFER?  APPLE3:SOROC.MISCINFO
TO WHERE?  SOROC1:SYSTEM.MISCINFO[1]
REMOVE OLD SOROC1:SYSTEM.MISCINFO ?  Y
APPLE3:SOROC.MISCINFO
 --> SOROC1:SYSTEM.MISCINFO
```

Note: the information you have just changed in making SOROC1: will not affect the system until you reboot the system with SOROC1: in the boot disk drive.

REMOVING LINEFEED FROM RETURN

Various printers used with the Apple Pascal system have different requirements for dealing with RETURN (carriage return, or ASCII CR) characters. Some printers require that a linefeed character follow every RETURN character, while other printers automatically supply their own linefeed following every RETURN character.

The file which contains the system's hardware configuration information, SYSTEM.MISCINFO, does not have any information about the printer's requirements. The Apple Pascal system normally sends out a linefeed after every RETURN character. This matches the requirements of most printers. However, on some printers this may cause double spacing between lines, and some printers are unable to work properly if sent these RETURN+linefeed combinations. For printers that do not work properly when sent a linefeed after every RETURN character, the Apple Pascal system provides the Linefeed utility program.

DISKFILES NEEDED

The following diskfile allows you to use the utility program for preventing a linefeed from being sent to the printer after every RETURN character.

LINEFEED.CODE	(any diskette, any drive; required only to start; changes memory only, no change to any permanent file)

The file LINEFEED.CODE is normally found on diskette APPLE3: . When the Linefeed utility program terminates, your boot diskette should be in the boot drive. If it is not there, the system will tell you to

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette).

One-drive note: You can first use the Filer to T(ransfer the file LINEFEED.CODE from APPLE3: onto your boot diskette. You are then ready to X(ecute LINEFEED with your boot diskette in the drive. This works well because the boot diskette (and the necessary file SYSTEM.PASCAL) is available when you return to COMMAND level after using the utility.

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive. You are then ready to X(ecute APPLE3:LINEFEED .

USING THE UTILITY

From command level, with APPLE3: in any available drive, type X for X(ecute. Answer the question

```
EXECUTE WHAT FILE?
```

by typing

```
APPLE3:LINEFEED
```

One-drive note: You have T(ransferred LINEFEED.CODE to your boot diskette, APPLE1: for example. You should therefore type

```
APPLE1:LINEFEED
```

(Note that you do not need to specify LINEFEED.CODE ; the .CODE suffix is automatically added if you forget to type it.) The system then executes LINEFEED.CODE . No messages are displayed, and the COMMAND prompt line reappears.

After running this utility, until the next system boot, RESET, or initialization, a linefeed is no longer sent to the printer after every RETURN character. This utility can be used to cure double spacing and other printer troubles associated with linefeeds.

EASIER USE OF THE UTILITY

If you use the same printer constantly, and it always needs to have no linefeed sent after RETURN characters, you will have to execute LINEFEED.CODE every time you start the system. You could T(ransfer LINEFEED.CODE permanently to your boot diskette, and that would certainly make things simpler, but you would still have to remember to X(ecute LINEFEED everytime you start the system.

Fortunately, there is an easier way: make yours a "turnkey" system, which automatically executes LINEFEED.CODE everytime you start the system. To do this, simply T(ransfer the file LINEFEED.CODE from APPLE3: to your boot diskette, and then C(hange its filename on the boot diskette to SYSTEM.STARTUP . That's all there is to it. Every time you boot your system using that boot diskette, the file SYSTEM.STARTUP will be automatically executed.

CALCULATOR

This utility turns your Apple into a VERY simple calculator capable of addition, subtraction, multiplication, and division only. Results are expressed in scientific notation, rounded to six digits and followed by a power-of-ten indicator from E37 to E-37. Input numbers must not be expressed in scientific notation, and should not contain more than 36 digits.

DISKFILES NEEDED

The following diskfile is needed in order to use the calculator utility program:

CALC.CODE	(any diskette, any drive; required only to start)

The file CALC.CODE is normally found on diskette APPLE3: . When you quit the calculator program, or when any error terminates the program, your boot diskette should be in the boot drive. If it is not there, the system may "hang" in more or less dramatic ways, or you may see the message

```
PUT IN APPLE1:
```

(if APPLE1: is your boot diskette). Even after correctly responding to this message, you will often have to re-initialize the system (by pressing the RESET key) before the system will respond normally to commands again.

One-drive note: On single-drive systems, the safest approach is to use the Filer to T(ransfer the file CALC.CODE from APPLE3: onto your boot diskette before X(ecuting CALC . This is best because the boot diskette (and the necessary file SYSTEM PASCAL) is available when you return to COMMAND level after quitting the utility or after any error terminates the program. However, you can start the calculator utility with APPLE3: in the drive. When the utility's first prompt appears, you should then put your boot diskette in the drive. Be sure the boot diskette is in the drive before you quit the calculator utility program, or your system may "hang".

Two-drive note: You will normally place your boot diskette in the boot drive, and place APPLE3: in the other drive. You are then ready to X(ecute APPLE3:CALC .

USING THE UTILITY

From Command level, with APPLE3: in any available disk drive, type X for X(ecute. When you see this question

```
EXECUTE WHAT FILE?
```

respond by typing

```
APPLE3:CALC
```

One-drive note: If you have T(ransferred CALC.CODE from APPLE3: to your boot diskette, as suggested, you will respond by typing (assuming APPLE1: is your boot diskette, for example)

```
APPLE1:CALC
```

(Note that you do not need to specify CALC.CODE ; the .CODE suffix is added automatically if you don't type it.) After this response, CALC.CODE is executed, and this prompt appears just below the top screen line saying EXECUTE WHAT FILE? :

```
->
```

You may now type any simple mathematical expression, using only these operators:

+	addition
-	subtraction
*	multiplication
/	division

All multiplications and divisions are carried out before additions and subtractions are executed. Use parentheses to keep portions of the expression unambiguous. When the expression is as you want it, press the RETURN key to see the result. Here are a few illustrative examples:

```
->(2+3)*4
     2.00000E1

->5/0
DIVISION BY ZERO: TRY AGAIN

->19.2357/2873.456
     6.69427E-3

->400.23+12.37+45.78-595.98+16.00
    -1.21600E2

->45*-2
"(" MISSING: TRY AGAIN

->45*(-2)
    -9.00000E1
```

Responding to the prompt -> by just pressing the RETURN key terminates the calculator utility and returns you to the operating system's Command level. Be sure your boot diskette is in the boot drive before you quit the program in this way.

Occasionally, you may run into a problem like this:

```
->3^2

UNIMPLEMENTED INSTRUCTION
S# 1, P# 10, I# 92
TYPE <SPACE> TO CONTINUE
```

After a message like this last one, you should make sure your boot diskette is in the boot drive, and then press the Apple's spacebar. This causes the system to be re-booted, and you can then X(ecute the calculator utility again.

UTILITIES SUMMARY

FORMATTING NEW DISKETTES

1. X(ecute APPLE3:FORMATTER
2. When asked FORMAT WHICH DISK? , put a new diskette in any drive and type that drive's volume number.
3. To quit, press the RETURN key in response to FORMAT WHICH DISK?

THE SYSTEM LIBRARIAN

1. X(ecute APPLE3:LIBRARY
2. When asked for an OUTPUT CODE FILE -> , type a filename for the new library file. E.g., MYDISK:NEW.LIBRARY
3. When asked for a LINK CODE FILE -> , type the name of the file which contains the first items to put in the new library. E.g., APPLE1:SYSTEM.LIBRARY
4. To transfer an item from the source Link Code File to the new library Output Code File, type the item's Link Code File slot number (0 to 15) and press the spacebar. When asked SLOT TO LINK INTO? , type the number of the slot you want the item to occupy in the Output Code File and press the spacebar.
5. Type N to begin taking items from a N(ew Link Code File.

6. When all desired items have been transferred to the new library, lock the new library by typing Q for Q(uit. When asked NOTICE? , type a copyright message or press RETURN.

7. To use the new library, it must be placed on your boot diskette and it must be named SYSTEM.LIBRARY

LIBRARY MAPPING

1. X(ecute APPLE3:LIBMAP

2. When prompted to ENTER LIBRARY NAME: , type the name of the library or other code file whose contents you wish to see mapped. E.g., APPLE1:SYSTEM.LIBRARY

3. When asked LIST LINKER INFO TABLE? , press the spacebar or RETURN key unless you want that information.

4. When prompted for a MAP OUTPUT FILE NAME: , type the name of the diskette file or other device to which you wish the map sent. Just pressing the RETURN key sends the map to CONSOLE: .

5. When prompted again to ENTER LIBRARY NAME: , type the name of the next library file whose contents you wish mapped, or press the RETURN key to quit the program.

SYSTEM RECONFIGURATION

1. To use your system with an external terminal, make a copy of APPLE1: for use as your new boot diskette. Give this new boot diskette a different name, such as SOROC1: or BRIAN: .

2. If your terminal is a Soroc IQ120, T(ransfer APPLE3:SOROC.MISCINFO to your new boot diskette and change its filename to SYSTEM.MISCINFO

   If your terminal is a Hazeltine 1500, T(ransfer APPLE3:HAZEL.MISCINFO to your new boot diskette and change its filename to SYSTEM.MISCINFO

   If your terminal is neither of the above, X(ecute APPLE3:SETUP , and let the program T(each you how to C(hange the parameters to suit your terminal. When the parameters are set correctly, Q(uit and do a D(isk update. This creates the file NEW.MISCINFO on your boot diskette. Then E(xit the program. Finally, you must T(ransfer NEW.MISCINFO to your new boot diskette and change this file's name to SYSTEM.MISCINFO

3. Read the next section, which tells how to change the GOTOXY Pascal procedure to work correctly with your terminal.

CHANGING GOTOXY COMMUNICATION

1. Read the previous section on reconfiguring your system to suit your external terminal.

2. X(ecute APPLE3:BINDER

3. When you are asked FILE WHICH CONTAINS GOTOXY? , type the name of the codefile containing the new GOTOXY procedure for your terminal. One-drive note: this file must be on your boot diskette.

   If your terminal is a Soroc IQ120, type APPLE3:SOROCGOTO

   If your terminal is a Hazeltine 1500, type APPLE3:HAZELGOTO

   If your terminal is neither of the above, type the name of the codefile containing a new GOTOXY procedure that you modified from APPLE3:SOROCGOTO.TEXT to suit your terminal and then compiled.

4. This program creates the file NEW.PASCAL on your boot diskette. You must now T(ransfer NEW.PASCAL onto the new boot diskette created in the previous section, and change the name of this file to SYSTEM.PASCAL . Your system will discover the new files SYSTEM.MISCINFO and SYSTEM.PASCAL the next time you boot with your new boot diskette.

REMOVING LINEFEED FROM RETURN

1. The Apple Pascal system automatically supplies a linefeed after every RETURN character sent to the printer. If your printer does not work correctly with this arrangement, X(ecute APPLE3:LINEFEED

2. Until the next boot, RESET, or initialization, no linefeed will be sent to the printer after RETURN characters.

3. You may wish to put LINEFEED.CODE on your boot diskette, and change its name to SYSTEM.STARTUP Then this utility will be executed automatically, each the system is booted.

CALCULATOR

1. X(ecute APPLE3:CALC

2. When you are prompted -> , type any mathematical expression involving decimal numbers with fewer than 36 digits, and the operators + - * / Use parentheses (and) to keep expressions unambiguous. Example:

```
->(2.32+.029)*(-1.75/4394.17)
```

3. Press the RETURN key to see the result of evaluating the expression, expressed in scientific notation with six digits and a power-of-ten indicator. Result of above example:

```
    -9.35501E-4
```

In other notation, -9.35501E-4 = $-9.35501*(10^{-4})$ = -.000935501

4. To quit, press RETURN when prompted ->

APPENDIX A

ARCHITECTURE OF THE P-MACHINE

224 TECHNICAL INFORMATION
224 Introduction
224 Hardware Emulation: Registers
225 Communication between Operating System and the P-Machine
226 Error Handling
227 Operand Formats
229 THE P-MACHINE INSTRUCTION SET
229 Instruction Formats
229 Conventions and Notation
230 One-Word Loads and Stores
  230 Constant
  231 Local
  231 Global
  231 Intermediate
  232 Indirect
  232 Extended
232 Multiple-Word Loads and Stores (sets and reals)
233 Byte Array Handling
233 String Handling
233 Record and Array Handling
234 Dynamic Variable Allocation
235 Top-of-Stack Arithmetic
  235 Integers
  236 Non-Integer Comparisons
  236 Reals
  237 Strings
  238 Logical
  238 Sets
  239 Byte Arrays
  239 Records and Word Arrays
239 Jumps
240 Procedure and Function Calls
242 System Support Procedures
  243 Byte-Array Procedures
  244 Compiler Procedures
  245 Miscellaneous

TECHNICAL INFORMATION

INTRODUCTION

The Apple Pascal "Pseudo-machine", or "P-machine", a version of the UCSD Pascal P-machine, is the software-generated "device" which executes P-code as its "machine" language. Every computer operating under a form of UCSD Pascal has been programmed to "look like" this common P-machine, from the viewpoint of a program being executed. The P-machine supports the following:

1. Variable addressing, including strings, byte arrays, packed fields, and dynamic variables

2. Logical, integer, real, set, array, and string top-of-stack arithmetic and comparisons

3. Multi-element structure comparisons

4. Several types of branches

5. Procedure and function calls and returns, including overlayable procedures

6. Miscellaneous procedures used by systems programs

This appendix, to be used in conjunction with the next appendix OPERATION OF THE P-MACHINE, describes the P-machine "hardware," communication with the operating system, error handling, and the mnemonic instruction set.

HARDWARE EMULATION: REGISTERS

The P-machine uses 16-bit words, with two 8-bit bytes per word. It has an evaluation stack, several registers, and a user memory containing a program stack and a heap. All registers are pointers to word-aligned structures, except IPC, which is a pointer to byte-aligned instructions. The registers, sometimes referred to as "pseudo-variables", are:

SP: evaluation Stack Pointer. A pointer to the current "top" of the evaluation stack (one byte beyond the last byte in use). In the Apple, the evaluation stack uses a portion of the 6502's hardware stack, starting in hex memory location 1FF and growing down toward hex location 100. It is used to pass parameters, return function values, and as an operand source for many instructions. The evaluation stack is extended by loads, and is cut back by stores and arithmetic operations.

IPC: Interpreter Program Counter. Contains the address of the next instruction to be executed, in the code segment of the currently executing procedure.

SEG: SEGment pointer points to the procedure dictionary of the segment to which the currently executing procedure belongs. (See this manual's appendix OPERATION OF THE P-MACHINE for illustrations.)

JTAB: Jump TABle pointer. A pointer to the table of attributes and jump table entries in the procedure code section of the currently executing procedure. (See this manual's appendix OPERATION OF THE P-MACHINE for illustrations.)

KP: program stacK Pointer. A pointer to the current top of the program stack. The program stack starts in high user memory and grows downward toward the heap. (See this manual's appendix OPERATION OF THE P-MACHINE for illustrations.)

MP: Markstack Pointer. A pointer to the low byte of MSSTAT, in the topmost Markstack on the program stack, in the activation record of the currently executing procedure. Variables local to the current procedure are accessed by indexing off MP.

NP: New Pointer. A pointer to the current top of the dynamic heap (one byte beyond the last byte in use). The heap starts in low user memory and grows upward toward the program stack. It contains all dynamic variables (see Jensen and Wirth, Chapter 10). It is extended by the standard procedure 'new', and is cut back by the standard procedure 'release'.

BASE: BASE Procedure. A pointer to the activation record of the most recently invoked base procedure (lex level 0). Global (lex level 0) variables are accessed by indexing off BASE.

COMMUNICATION BETWEEN OPERATING SYSTEM AND THE P-MACHINE

It is sometimes necessary for the operating system and the P-machine to exchange information. Hence there exists a variable SYSCOM in the outer block of the operating system, and a corresponding area in memory known to the P-machine. The fields in SYSCOM actually relevant to this communication are:

IORSLT: Contains the error code returned by the last activated or terminated I/O operation (see I/O Error Messages in this manual's TABLES appendix, and the Apple Pascal Language Reference Manual's description of Apple Pascal's read and write procedures).

XEQERR: Contains the error code of the last execution error (see the Error Handling section in this appendix, and Execution Error Messages in the TABLES appendix).

SYSUNIT: Contains the volume number of the device from which the operating system was booted (usually the boot disk drive, volume 4).

BUGSTATE: (Not used; intended for future debugging routines.)

GDIRP: Contains a pointer to the most recent disk directory read in, unless dynamic allocation or deallocation has taken place since then (see the MRK, RLS, and NEW instructions).

STKBASE, LASTMP, SEG, JTAB: Contains copies of the BASE, MP, SEG and JTAB registers.

BOMBP: Contains a pointer to the activation record of the operating system routine EXECERROR when an execution error occurs (see the Error Handling section of this appendix).

BOMIPC: Contains the value of IPC when an execution error occurs.

HLTLINE: (Not used; intended for future debugging routines.)

BRKPTS: (Not used; intended for future debugging routines.)

CRTINFO.EOF: Contains the end-of-file character (see discussion of the reconfiguration program in this manual's chapter UTILITY PROGRAMS).

CRTINFO.FLUSH: Contains the flush-output character (see the discussion of the reconfiguration program in the chapter UTILITY PROGRAMS).

CRTINFO.STOP: Contains the stop-output character (see discussion of the reconfiguration program in this manual's chapter UTILITY PROGRAMS).

CRTINFO.BREAK: Contains the break-execution character (see discussion of the reconfiguration program in the chapter UTILITY PROGRAMS).

SEGTABLE: Contains the segment dictionary for the operating system (segments 0, and 2 through 6) and for the currently executing system or user program (segment 1: main program; 7 through 21: segment procedures and regular Units; 22 through 31: Intrinsic Units) (see the appendix OPERATION OF THE P-MACHINE for illustrations).

ERROR HANDLING

Whenever an execution error occurs, the P-machine stops executing the current instruction (ideally leaving the evaluation stack in as nice a condition as possible) and transfers control to the interpreter's XEQERR routine. This routine does the following:

1. Enters the error code into SYSCOM^.XEQERR ,
2. Calculates what MP will be after step 4, and sets SYSCOM^.BOMBP to that (the size of EXECERROR's activation record must be known by the P-machine),
3. Stores the current value of IPC into SYSCOM^.BOMIPC ,
4. Points IPC to a CXP 0,2 P-code instruction (call operating system procedure EXECERROR) and
5. Resumes execution of interpreter code, starting with the CXP .

OPERAND FORMATS

Although an element of a structure may occupy as little as one bit, as in a PACKED ARRAY OF boolean, variables in the P-machine are always aligned on word boundaries. Words consist of two bytes of which the even-address byte is least significant. All top-of-stack operations expect their operands to occupy at least one word on the evaluation stack, even if not all the information in a word is valid. The least significant bit of a word is bit 0, the most significant is bit 15.

BOOLEAN: One word. Bit 0 indicates the value (false=0, true=1), and this is the only information used by boolean comparisons. However, the boolean operators LAND, LOR, and LNOT operate on all 16 bits.

INTEGER: One word, two's complement, capable of representing values in the range -32768..32767.

LONG INTEGER: 3..11 words. A variable declared as INTEGER[n] is allocated ((n+3) DIV 4) + 2 words. Regardless of the value of the integer, its actual size agrees with its allocated size. Each decimal digit of a long integer is stored as four bits of Binary Coded Decimal. The format of long integers is as follows:

word n (tos-n):	high byte contains the two least significant decimal digits (BCD).
" " "	" " " " " " " "
" " "	" " " " " " " "
word 2 (tos-2):	low byte contains the two most significant decimal digits (BCD).
word 1 (tos-1):	low byte contains the sign (all zeros = plus, all ones = minus); high byte not used.
word 0 (tos-0):	contains the allocated length, in words.

SCALAR (user-defined): One word, in range 0..32767.

CHAR: One word, with low byte containing character. The internal character set is "extended" ASCII, with 0..127 representing the standard ASCII set, and 128..255 as a user-defined character set.

REAL: Two words, whose format is implementation dependent. The system is arranged so that only the interpreter needs to know the detailed internal format of REALs (beyond the fact that they occupy two words). In general, the format for 32-bit real numbers is as follows:

	Word 1 (tos)			Word 0 (tos)
Bit:	31	30...23	22..16 and	15..0
Item:	Sign	Exponent	Mantissa	

POINTER: One or three words, depending on type of pointer.

Pascal pointers, internal word pointers: one word, containing a word address.

Internal byte pointers: one word, containing a byte address.

Internal packed field pointers: three words.
word 2 (Tos-2): word pointer to word field is in.
word 1 (Tos-1): field width (in bits).
word 0 (Tos-0): right bit number of field.

SET: 0..31 words in activation record, 1..32 words on evaluation stack. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as SET OF m..n is allocated (n+15) DIV 16 words. When a set is in the activation record, all words allocated contain valid information (the set's actual size agrees with its allocated size).

When a set is on the evaluation stack, it is represented by a word containing the length (tos), and then that number of words of information. The set may be padded with extra words (to compare it with another set of different size, say), the length word indicating the number of words in the structure padded. Before being stored back in the activation record, a set must be forced back to the size allocated to it, by issuing an ADJ instruction.

RECORDS and ARRAYS: Any number of words. Arrays are stored in forward order, with higher-indexed array elements appearing in higher-numbered memory locations. Only the address of the record or array is loaded onto the evaluation stack, never the structure itself. Packed arrays must have an integral number of elements in each word, as there is no packing across word boundaries (it is acceptable to have unused bits in each word). The first element in each word has bit 0 as its low-order bit.

STRINGS: 1..128 words. Strings are a flexible version of PACKED ARRAYs OF char. A string[n] occupies (n DIV 2)+1 words. Byte 0 of a string is the current length of the string, and bytes 1..length(string) contain valid characters.

CONSTANTS: Constant scalars, sets, and strings may be imbedded in the instruction stream, in which case they have special formats.

All scalars (excluding reals) greater than 127: Two bytes, high byte first.

Strings: All string literals take length(literal)+1 bytes, and are word aligned. The first byte is the length, the rest are the actual characters. This format applies even if the literal should be interpreted as a PACKED ARRAY OF CHAR.

Reals and sets: Word aligned, and in REVERSE word order.

THE P-MACHINE INSTRUCTION SET

INSTRUCTION FORMATS

Instructions on the P-machine are one or two bytes long, followed by zero to four parameters. Most parameters specify one word of information, and are one of five basic types:

UB: Unsigned Byte. High order byte of parameter is implicitly zero.

SB: Signed Byte. High order byte is sign extension of bit 7.

DB: Don't-care Byte. Can be treated as SB or UB, as value is always in the range 0..127.

B: Big. This parameter is one byte long when used to represent values in the range 0..127, and is two bytes long when representing values in the range 128..32767. If the first byte is in 0..127, the high byte of the parameter is implicitly zero. Otherwise, bit 7 of the first byte is cleared and it is used as the high order byte of the parameter. The second byte is used as the low order byte.

W: Word. The next two bytes, low byte first, give the parameter value.

Any exceptions to these formats are noted in the instruction descriptions.

CONVENTIONS AND NOTATION

The program stack, which starts at user high memory and grows downward, contains program code segments and activation records for currently active procedures, and data segments associated with INTRINSIC UNITs.

The evaluation stack, which starts at hex location 1FF and grows downward toward hex 100, contains operands and other temporary items needed during expression evaluation. When an instruction is said to "push" an item, that item is placed on the top of the evaluation stack (remember that the evaluation stack grows downward).

In referring to operands on the evaluation stack (for example, tos or tos-1), each operand can contain from one word to 256 words, depending on the context. Also, unless specifically noted to the contrary, operands used by an instruction are popped off the evaluation stack (removed from the stack and not put back there) as they are used.

Abbreviations are used widely, but use fairly simple conventions. Parameters are written as X or Xn , where X is UB , SB , DB , B , or W , and n is an integer indicating the parameter position in the instruction (used in the descriptions to differentiate between several parameters that would otherwise have the same name). The term tos means the operand on the top of the evaluation stack, tos-1 is the next operand, etc. The Mark Stack Control Word, or MSCW, is simply called the Markstack.

Many instructions refer to the activation record of a procedure, and this appendix assumes the reader has a general knowledge of procedure-calling in stack machines, and the concept of stack frames. An activation record as defined in this appendix specifically consists of:

1) the local variables for the procedure,

2) parameters passed to the procedure at the time of its invocation,

3) space for storing the value returned by the procedure, if the procedure is a function, and

4) the Markstack, containing addressing information (Static Links), and information on the calling procedure's environment when the procedure was called (see this manual's appendix, OPERATION OF THE P-MACHINE, for illustrations).

The dynamic chain refers to the calling chain, traversed using the Markstack Dynamic Links (MSCW.MSDYN). The static chain refers to the lexical or ancestor chain, traversed using the Markstack Static Links (MSCW.MSSTAT).

The columns of information in the various instruction descriptions may be labelled as follows:

Column 1	Column 2	Column 3	Column 4
Op-Code Mnemonic	Decimal Op-Code	Instruction Parameters	Full Name and Operation of the Instruction

ONE-WORD LOADS AND STORES

Constant

SLDC 0	0		Short load one-word constant. For an instruction SLDC x , push the opcode, x , with high byte zero.
SLDC 1	1		
: :	:		
SLDC 127	127		
LDCN	159		Load constant NIL. Push the implementation-dependent value of NIL (0, on the Apple).
LDCI	199	W	Load one-word constant. Push W.

Local

SLDL 1	216		Short load local word. For an instruction SLDL x , fetch the word with offset x in MP activation record and push it.
SLDL 2	217		
: :	:		
SLDL 16	231		
LDL	202	B	Load local word. Fetch the word with offset B in MP activation record and push it.
LLA	198	B	Load local address. Fetch address of the word with offset B in MP activation record and push it.
STL	204	B	Store local word. Store tos into word with offset B in MP activation record.

Global

SLDO 1	232		Short load global word. For an instruction SLDO x , fetch the word with offset x in BASE activation record and push it.
SLDO 2	233		
: :	:		
SLDO 16	247		
LDO	169	B	Load global word. Fetch the word with offset B in BASE activation record and push it.
LAO	165	B	Load global address. Fetch address of word with offset B in BASE activation record and push it.
SRO	171	B	Store global word. Store tos into word with offset B in BASE activation record.

Intermediate

LOD	182	DB,B	Load intermediate word. Fetch word with offset B in the activation record found by traversing DB Static Links, and push it.
LDA	178	DB,B	Load intermediate address. Fetch address of word with offset B in the activation record found by traversing DB Static Links, and push it.
STR	184	DB,B	Store intermediate word. Store tos into the word with offset B in activation record found by traversing DB Static Links.

Indirect

SIND 0	248		Load indirect word. Fetch the word pointed to by tos and push it (this is a special case of SIND x , described below).
SIND 1 SIND 2 : : SIND 7	249 250 : 255		Short index and load word. For an instruction SIND x , index the word pointer tos by x words, and push the word pointed to by the result.
IND	163	B	Static index and load word. Index the word pointer tos by B words, and push the word pointed to by the result.
STO	154		Store indirect word. Store tos into the word pointed to by tos-1 .

Extended

LDE	157	UB,B	Load extended word. Fetch the word with offset B in data segment UB (from an Intrinsic Unit) and push it.
LAE	167	UB,B	Load extended address. Fetch address of the word with offset B in data segment UB (from an Intrinsic Unit), and push it.
STE	209	UB,B	Store extended word. Store tos into the word with offset B in data segment UB (from an Intrinsic Unit).

MULTIPLE-WORD LOADS AND STORES (REALS AND SETS)

LDC	179	UB,<block>	Load multiple-word constant. Fetch word-aligned <block> of UB words in reverse word order, and push the block.
LDM	188	UB	Load multiple words. Fetch a block of UB words, whose beginning is pointed to by tos , and push the block.
STM	189	UB	Store multiple words. Tos is a block of UB words, tos-1 is a word pointer to a similiar block. Transfer the block from tos to the destination block pointed at by tos-1.

BYTE ARRAY HANDLING

LDB	190		Load byte. Index the byte pointer tos-1 by the integer index tos , and push the byte (after zeroing high byte) pointed to by the resulting byte pointer.
STB	191		Store byte. Index the byte pointer tos-2 by the integer index tos-1 , and push the byte tos into the location pointed to by the resulting byte pointer.

STRING HANDLING

LSA	166	UB,<chars>	Load constant string address. Push a byte pointer to the location containing UB, and then skip IPC past <chars>.
SAS	170	UB	String assign. Tos is either a source byte pointer or a character. (Characters always have a high byte of zero, while pointers never do.) Tos-1 is a destination byte pointer. UB is the declared size of the destination string. If the declared size is less than the current size of the source string, give an execution error; otherwise transfer all bytes of source containing valid information to the destination string.
IXS	155		Index string array. Index the byte pointer tos-1 by the integer index tos , and push the resulting byte pointer if it is in the range 1..current length. If not, give an execution error.

RECORD AND ARRAY HANDLING

MOV	168	B	Move words. Transfer a source block of B words, pointed to by byte pointer tos , to a similar destination block pointed to by byte pointer tos-1 .
INC	162	B	Increment field pointer. Index the word pointer tos by B words and push the resultant word pointer.

IXA	164	B	Index array. Tos is an integer index, tos-1 is the array base word pointer, and B is the size (in words) of an array element. Compute a word pointer to the indexed element and push the pointer.
IXP	192	UB1,UB2	Index packed array. Tos is an integer index, tos-1 is the array base word pointer. UB1 is the number of elements per word, and UB2 is the field width (in bits). Compute a packed field pointer to the indexed field and push the resulting pointer.
LPA	208	UB,<chars>	Load a packed array. Push a byte pointer to the first location following the one that contains UB , and then skip IPC past <chars>.
LDP	186		Load a packed field. Fetch the field indicated by the packed field pointer tos , and push it.
STP	187		Store into a packed field. Store the data tos into the field indicated by the packed field pointer tos-1 .

DYNAMIC VARIABLE ALLOCATION

NEW	158 1	New variable allocation. Tos is the size (in words) to allocate the variable, and tos-1 is a word pointer to a dynamic variable. GDIRP is a pointer to a temporary directory buffer placed in memory directly above the heap. If GDIRP is non-NIL, set GDIRP to NIL. Store NP into the word pointed to by tos-1 , and increment NP by tos words.
MRK	158 31	Mark heap. Set GDIRP to NIL if necessary; then store NP into the word indicated by word pointer tos .
RLS	158 32	Release heap. Set GDIRP to NIL, then store the word indicated by the word pointer tos into NP .

TOP-OF-STACK ARITHMETIC

Integers

Note: Overflows do not cause an execution error.

ABI	128	Absolute value of integer. Push the absolute value of integer tos . Result is undefined if tos is initially -32768.
ADI	130	Add integers. Add tos and tos-1 , and push the resulting sum.
NGI	145	Negate integer. Push the two's complement of tos .
SBI	149	Subtract integers. Subtract tos from tos-1 , and push the resulting difference.
MPI	143	Multiply integers. Multiply tos and tos-1 , and push the resulting product. This instruction may cause an overflow if the result is larger than 16 bits.
SQI	152	Square integer. Square tos , and push the result. May cause overflow if result is larger than 16 bits.
DVI	134	Divide integers. Divide tos-1 by tos and push the resulting integer quotient (any remainder is discarded).
MODI	142	Modulo integers. Divide tos-1 by tos and push the resulting remainder (as defined in Jensen and Wirth).
CHK	136	Check against subrange bounds. Insure that tos-1 <= tos-2 <= tos , leaving tos-2 on the stack. If conditions are not satisfied, give an execution error.
EQUI	195	Tos-1 = tos .
NEQI	203	Tos-1 <> tos .
LEQI	200	Tos-1 <= tos .
LESI	201	Tos-1 < tos .
GEQI	196	Tos-1 >= tos .
GRTI	197	Tos-1 > tos .

Integer comparisons. Compare tos-1 to tos and push the result, TRUE or FALSE.

Non-Integer Comparisons

EQU	175	UB	Tos-1 = tos .
NEQ	183	UB	Tos-1 <> tos .
LEQ	180	UB	Tos-1 <= tos .
LES	181	UB	Tos-1 < tos .
GEQ	176	UB	Tos-1 >= tos .
GRT	177	UB	Tos-1 > tos .

Compare tos-1 to tos , and push the result, TRUE or FALSE. The type of comparison is specified by UB :

Contents of Tos-1 & tos	Value of UB for comparison
reals	2
strings	4
booleans	6
sets	8
byte arrays	10
words	12

Comparisons using specific values of UB are shown in the following instruction descriptions.

Reals

Note: All over/underflows cause an execution error.

FLT	138	Float top-of-stack. Convert the integer tos to a floating point number, and push the result.
FLO	137	Float next to top-of-stack. Tos is a real, tos-1 is an integer. Convert tos-1 to a real number, and push the result.
TNC	158 22	Truncate real. Truncate (as defined in Jensen and Wirth) the real tos and convert to an integer, and then push the result.
RND	158 23	Round real. Round (as defined in Jensen and Wirth) the real tos , then truncate and convert to an integer, and finally push the result.
ABR	129	Absolute value of real. Push the absolute value of the real tos .

ADR	131	Add reals. Add tos and tos-1 , and push the resulting sum.
NGR	146	Negate real. Negate the real tos , and push the result.
SBR	150	Subtract reals. Subtract tos from tos-1 , and push the resulting remainder.
MPR	144	Multiply reals. Multiply tos and tos-1 , and push the resulting product.
SQR	153	Square real. Square tos , and push the result.
DVR	135	Divide reals. Divide tos-1 by tos , and push the resulting quotient.
POT	158 35	Power of ten. If the integer tos is in the range 0 <= tos <= 38 , push the real (and thus implementation-dependent) value 10 ^ tos . If not, give an execution error. This facility allows the rest of the system to be independent of floating point format.

EQUREAL	175	2	Tos-1 = tos .
NEQREAL	183	2	Tos-1 <> tos .
LEQREAL	180	2	Tos-1 <= tos .
LESREAL	181	2	Tos-1 < tos .
GEQREAL	176	2	Tos-1 >= tos .
GTRREAL	177	2	Tos-1 > tos .

Real comparisons. Compare the real tos-1 to the real tos , and push the result, TRUE or FALSE.

Strings

EQUSTR	175	4	Tos-1 = tos .
NEQSTR	183	4	Tos-1 <> tos .
LEQSTR	180	4	Tos-1 <= tos .
LESSTR	181	4	Tos-1 < tos .
GEQSTR	176	4	Tos-1 >= tos .
GRTSTR	177	4	Tos-1 > tos .

String comparisons. Find the string pointed to by word pointer tos-1 , compare it lexicographically to the string pointed to by word pointer tos , and push the result, TRUE or FALSE.

Logical

LAND	132		Logical and. Push the result of tos-1 AND tos .
LOR	141		Logical or. Push the result of tos-1 OR tos .
LNOT	147		Logical not. Push the one's complement of tos .
EQUBOOL	175	6	Tos-1 = tos .
NEQBOOL	183	6	Tos-1 <> tos .
LEQBOOL	180	6	Tos-1 <= tos .
LESBOOL	181	6	Tos-1 < tos .
GEQBOOL	176	6	Tos-1 >= tos .
GRTBOOL	177	6	Tos-1 > tos .

Boolean comparisons. Compare bit 0 of tos-1 to bit 0 of tos and push the result, TRUE or FALSE.

Sets

ADJ	160	UB	Adjust set. Force the set tos to occupy UB words, either by expansion (putting zeroes "between" tos and tos-1) or compression (chopping of high words of set), discard the length word, and push the resulting set.
SGS	151		Build a singleton set. If the integer tos is in the range 0 <= tos <= 511 , push the set [tos] . If not, give an execution error.
SRS	148		Build a subrange set. If the integer tos is in the range 0 <= tos <= 511 , and the integer tos-1 is in the same range, push the set [tos-1..tos] (push the set [] if tos-1 > tos). If either integer exceeds the range, give an execution error.
INN	139		Set membership. If integer tos-1 is in set tos , push TRUE. If not, push FALSE.
UNI	156		Set union. Push the union of sets tos and tos-1 . (tos OR tos-1)
INT	140		Set intersection. Push the intersection of sets tos and tos-1 . (tos AND tos-1)

DIF	133		Set difference. Push the difference of sets tos-1 and tos . (tos-1 AND NOT tos).

EQUPOWR	175	8	Tos-1 = tos .
NEQPOWR	183	8	Tos-1 <> tos .
LEQPOWR	180	8	Tos-1 <= (subset of) tos .
GEQPOWR	176	8	Tos-1 >= (superset of) tos .

Set comparisons. Compare set tos-1 to the set tos , and push the result, TRUE or FALSE.

Byte Arrays

EQUBYT	175	10 , B	Tos-1 = tos .
NEQBYT	183	10 , B	Tos-1 <> tos .
LEQBYT	180	10 , B	Tos-1 <= tos .
LESBYT	181	10 , B	Tos-1 < tos .
GEQBYT	176	10 , B	Tos-1 >= tos .
GRTBYT	177	10 , B	Tos-1 > tos .

Byte array comparisons. Compare byte array tos-1 to byte array tos , and push the result, TRUE or FALSE. <=, <, >=, and > must be used with PACKED ARRAYs OF CHAR, only. B gives the number of bytes to compare.

Records and Word Arrays

EQUWORD	175	12 , B	Tos-1 = tos .
NEQWORD	183	12 , B	Tos-1 <> tos .

Word or multiword structure comparisons. Compare word structure tos-1 to word structure tos , and push the result, TRUE or FALSE. B gives the number of bytes to compare.

JUMPS

Simple (non-case statement) jumps are all two bytes long. The first byte is the op-code, the second is a SB jump offset. If this offset is non-negative, it is simply added to IPC. (A value of zero for the jump offset will make any jump a two-byte NOP.) If SB is negative, then SB DIV 2 is used as a word offset into JTAB, and IPC is set to the byte address (JTAB^[SB DIV 2]) - contents of (JTAB[SB DIV 2]).

UJP	185	SB	Unconditional jump. Jump as described above.
FJP	161	SB	False jump. Jump if tos is FALSE.

EFJ	211	SB	Equal false jump. Jump if integer tos-1 <> integer tos .
NFJ	212	SB	Not equal false jump. Jump if integer tos-1 = integer tos .
XJP	172	W1,W2,W3, <case table>	Case jump. W1 is word-aligned, and is the minimum index of the table. W2 is the maximum index. W3 is an unconditional jump instruction past the table. The case table is (W2 - W1 + 1) words long, and contains self-relative locations. If tos , the actual index, is not in the range W1..W2 , then point IPC at W3 . Otherwise, use (tos - W1) as an index into the case table, and set IPC to the byte address (casetable[tos-W1]) minus the contents of (casetable[tos-W1]).

PROCEDURE AND FUNCTION CALLS

The general scheme used in procedure/function invocation is

1) Find the procedure code section for the called procedure. From the table of attributes (JTAB) in the called procedure's code section, find the data size and parameter size of the called procedure (for more details, see this manual's appendix, OPERATION OF THE P-MACHINE).

2) Extend the program stack by a number of bytes equal to the data size plus the parameter size.

3) Copy a number of bytes equal to the parameter size, from the evaluation stack's tos (pointed to by SP) to the beginning of the space just allocated. This passes parameters to the new procedure from its caller.

4) Build a Markstack, saving SP, IPC, SEG, JTAB, MP, and a Static Link pointer to the most recent activation record of the called procedure's immediate parent.

5) Calculate new values for SP, IPC, JTAB, MP, and if necessary, SEG. Check for program stack overflow.

6) If the called procedure has a lex level of -1 or 0 save BASE on the evaluation stack and calculate a new BASE.

7) Save KP on the program stack and calculate a new KP.

CLP	206	UB	Call local procedure. Call procedure UB , which is an immediate child of the currently executing procedure and in the same segment. Static Link of Markstack is set to old MP.
CGP	207	UB	Call global procedure. Call procedure UB , which is at lex level 1 and in the same segment as the currently executing procedure. Static Link of the Markstack is set to BASE.
CIP	174	UB	Call intermediate procedure. Call procedure UB in same segment as the currently executing procedure. The Static Link of the Markstack is set by looking up the call chain until an activation record is found whose caller had a lex level one less than the procedure being called. Use that activation record's Static Link as the Static Link of the new Markstack.
CBP	194	UB	Call base procedure. Call procedure UB , which is at lex level -1 or 0. The Static Link of the Markstack is set to the Static Link in BASE's activation record. The BASE is saved, after which it is pointed at the activation record just created.
CXP	205	UB1,UB2	Call external procedure. Call procedure UB2 , in segment UB1 . Used to call any procedure not in the same segment as the calling procedure, including procedures at lex level -1 or 0. It works as follows: 1) Is desired segment in memory? 2a) No: read in segment from disk using the information in the SEGTABLE, then build an activation record. 2b) Yes: build activation record normally. 3) Calculate the Static Link for the Markstack: if the called procedure has a lex level of -1 or 0, set as in CBP; otherwise set as in CIP.

CSP	158	UB	Call standard procedure. Call the standard Pascal procedure UB , where UB is used as an index into the CSP table in the interpreter. All instructions with decimal op-code 158 are examples of procedure calls using the CSP instruction.
RNP	173	DB	Return from non-base procedure. DB is the number of words that should be returned as a function value (0 for procedures, 1 for non-real functions, and 2 for real functions). Copy DB words from the bottom of the current procedure's activation record, and push them onto the evaluation stack. Then use the information in the current Markstack to restore the calling procedure's correct environment.
RBP	193	DB	Return from base procedure. Move the saved base into BASE, and then proceed as in the RNP instruction.
EXIT	158 4		Exit from procedure. Tos is the procedure number, tos-1 is the segment number. First, set IPC to point to the exit code of the currently executing procedure. Then, if the current procedure is the one to exit from, return control to the instruction fetch loop. Otherwise, change the IPC of each Markstack to point to the exit code of the procedure that invoked it, until the desired procedure is found. If at any time the saved IPC of main body of the operating system is about to be changed, give an execution error.

SYSTEM SUPPORT PROCEDURES

Note: See the Apple Pascal Language Reference Manual for a more detailed description of these procedures.

Byte-Array Procedures

FLC	158 10	Fillchar(dst, len, char). Tos (char) is the source character. Tos-1 (len) is the number of bytes in the destination array which are to be filled with the source char. Tos-2 (dst) is a byte pointer to the first byte to be filled in the destination PACKED ARRAY OF CHARacters. Copy the character from tos into tos-1 characters of the destination array.
SCN	158 11	Scan(maxdisp, mask, char, start, forpast). Tos (forpast) is a two-byte quantity (usually the default integer 0) which is pushed, but later discarded without being used in this implementation. Tos-1 (start) is a byte pointer to the first character to be scanned in a PACKED ARRAY OF CHARacters. Tos-2 (char) is the character against which each scanned character of the array is to be checked. Tos-3 (mask) is 0 if the check is for equality, or 1 if the check is for inequality. Tos-4 (maxdisplacement) gives the maximum number of characters to be scanned (scan to the left if negative). If a character check yields TRUE, push the number of characters scanned (negative, if scanning to the left). If maxdisp is reached before character check yields TRUE, push maxdisp .
MVL	158 02	Moveleft(src, dst, numbytes). Tos (numbytes) gives the number of bytes to move. Tos-1 (dst) is a byte pointer to the destination array's first byte which will receive a moved byte. Tos-2 (src) is a byte pointer to the source array's first byte which will be moved. Copy tos bytes from the source array to the destination array, proceeding to the right through both arrays.
MVR	158 03	Moveright(src, dst, numbytes). Tos (numbytes) gives the number of bytes to move. Tos-1 (dst) is a byte pointer to the destination array's first byte which will receive a moved byte. Tos-2 (src) is a byte pointer to the source array's first byte which will be moved. Copy tos bytes from the source array to the destination array, proceeding to the left through both arrays.

Compiler Procedures

BPT 213 B

Breakpoint. Not used (acts as a NOP); intended for future debugging routines.

TRS 158 08

Treesearch(fcp, fcp2, name). Tos-2 (fcp) is a byte pointer to the root of a binary tree. Tos (name) is a byte pointer to a location which contains the address of an 8-character name that you wish to find or to place in the tree. Search the tree, looking for a record with the required name. Store the address of the last node visited, on completion of the search, into the location pointed to by byte pointer tos-1 (fcp2), and push the result of the search:

- 0 if the last node was a record with the search name,
- 1 if the search name should be a new record, attaching to the last tree node by the Right Link,
- -1 if the search name should be a new record, attaching to the last tree node by the Left Link.

This is an assembly-language binary tree search used by the Compiler. It is fast, but does NOT do type checking on the parameters. The binary tree uses nodes of type

```
CTP =  RECORD
         NAME: PACKED ARRAY [1..8] OF CHAR;
         LLINK, RLINK: ^CTP;
         ...
          other information
         ...
         END;
```

IDS 158 07

Idsearch. Used by the Compiler to parse reserved words and identifiers.

Miscellaneous

TIM	158 09	Time. Pop two pointers to two integers, and place in those integers a 32-bit indication of the current time. (Since the Apple has no real-time clock, this instruction acts as a NOP. Apple sets both integers to zero.)
XIT	214	Exit the operating system. Do a "cold boot" of the system, like the operating system's H(alt command.
NOP	215	No operation. Sometimes used to reserve space in the code for later additions.

APPENDIX B

OPERATION OF THE P-MACHINE

248 INTRODUCTION
248 THE SYSTEM CODEFILE
248 Segments
249 A Codefile on Diskette
250 The Segment Dictionary
250 A Code Segment
251 A Procedure Dictionary
252 A Procedure Code Section
254 SYSTEM MEMORY USE
254 Apple II Memory Map
256 The Program Stack
258 An Activation Record
260 More on the Program Stack
260 OVERVIEW
260 Summary of the Figures
263 "The Program"

INTRODUCTION

The Apple Pascal system is a version of the UCSD Pascal system, which is an interpreter-based implementation of Pascal. This means that the Compiler emits code for a "Pseudo-machine" or "P-machine" which is emulated at run time by a program written in the machine language of the host. For the Apple, this P-machine emulation program is the interpreter, written in the Apple's 6502 machine language and found in the boot diskette's file SYSTEM.APPLE .

The Apple Pascal operating system and various utilities are themselves written in Pascal and run on the same interpreter. Thus the entire system can be moved to a new host machine by rewriting the interpreter for the new host. Every host computer operating under a version of UCSD Pascal has an interpreter that makes the host computer "appear", from the viewpoint of a program being executed, to be this same P-machine. This appendix describes the "run-time" or "execution-time" environment of the Apple Pascal P-machine. For more information about the "hardware" of the P-machine, see this manual's previous appendix, ARCHITECTURE OF THE P-MACHINE.

At the end of this appendix is a skeleton version of a large Pascal program, referred to throughout the appendix as "The Program". The main body of this appendix is a top-down description of "The Program"'s diskette codefile, and its execution under the Apple Pascal system.

We will make occasional use of a helpful coincidence: "The Program" briefly sketches out a very early version of the Apple Pascal operating system. The current Apple Pascal operating system has been extended and changed in many ways from "The Program" shown in the examples. However, this will not prevent you from understanding the mechanisms of the P-machine's operation, which are accurately described.

THE SYSTEM CODEFILE

SEGMENTS

If "The Program" were expanded to the complete Apple Pascal operating system, it would consist of at least 10,000 lines of Pascal and compile to more than 100,000 bytes of code -- just a bit too big to fit all at once into the memory of an Apple. Therefore, "The Program" is overlayed using "segments", which let you explicitly partition a program into portions, only some of which need be resident in main memory at a time. See the Apple Pascal Language Reference Manual for details about segments.

Segments used within a program portion must be declared before the body of the outer program portion. To let an inner segment call an outer-program procedure, the outer program portion can declare the called procedure FORWARD before declaring the segment. An example of this appears in "The Program", where the segment procedure COMPILER uses the outer program's procedure CLEARSCREEN, which was declared FORWARD.

Segmenting a program does not change its meaning in any fundamental sense. When a segment procedure is called (for instance, line A of "The Program" calls the COMPILER segment procedure), the interpreter checks to see if that segment is already on the program stack, due to a previous (and still active) invocation of the segment. If it is, control is transferred and execution proceeds; if not, the appropriate code segment must be loaded onto the program stack from disk before the transfer of control takes place. When no more active invocations of the segment exist, its code is removed from the program stack. For instance, in "The Program", the code for the COMPINIT segment is not present on the stack either before or after the execution of line A. In fact, the COMPINIT segment is only present on the stack during the execution of "The Program"'s line B.

A CODEFILE ON DISKETTE

The diskette codefile resulting from compilation of "The Program" is diagrammed in Figure 1. The codefile consists of a segment dictionary followed by a sequence of code segments. The main program generates one code segment, and each segment procedure generates another code segment. The ordering of code segments (from low address to high address) is determined by the order that one encounters segment procedure bodies in passing through "The Program".

```
                 high diskette addresses

      |====================================|   ---|
      | Segment #0          PASCALSYSTEM   |      |
      |------------------------------------|      |
      | Segment #4              EDITOR     |      |
      |------------------------------------|      | Code
      | Segment #3             COMPILER    |      | Segments
      |------------------------------------|      |
      | Segment #2             COMPINIT    |      |
      |------------------------------------|      |
      | Segment #1          USER PROGRAM   |      |
      |====================================|   ---|
      |         SEGMENT DICTIONARY         |   <---- Segment
      |====================================|         Dictionary

                 low diskette addresses
```

FIGURE 1   COMPLETE CODEFILE OF "THE PROGRAM"

Each code segment begins on a boundary between diskette blocks (the 512-byte disk allocation quantum used by the Apple Pascal operating system). Each segment may occupy many, many blocks; the code for these segments is only hinted at in the much-abbreviated version shown in "The Program".

> * An overview of the relationship between Figures 1 through 7 (to be discussed in the following pages) is given in Figure 8 at the end of this appendix. It is helpful to study Figure 8 at this point for a better understanding of the following sections.

THE SEGMENT DICTIONARY

The segment dictionary, in the first block of a codefile, contains an entry for each code segment in the file (the main program is assigned segment #0). The entry includes the segment's size (in bytes) and its disk location. The disk location is given as the number of blocks to the beginning of the code segment, relative to the beginning of the segment dictionary (which is also the beginning of the codefile). This information is kept in the system communications area (also called SYSCOM; see this manual's appendix, ARCHITECTURE OF THE P-MACHINE) during the execution of the codefile, and is used in the loading of non-present segments when they are needed. The segment dictionary also contains information about the code and data segments of INTRINSIC UNITs which the program USES. This manual's appendix, FILE FORMATS, gives a more detailed account of the information in the segment dictionary.

A CODE SEGMENT

Figure 2 is a detailed diagram of "The Program"'s Segment #0, containing the code for the main program segment, PASCALSYSTEM. Each code segment contains the code for that segment's outer block, as well as the code for each of the (non-segment) procedures within that segment. Observe that CLEARSCREEN is the first main-program procedure for which code is generated and that it appears at the beginning of the segment. The outer block code, which is generated last, appears last in the code segment. Following the code for the various procedures is the code segment's procedure dictionary.

```
                      high diskette addresses

           |==========================================|   --|
           | Number of procedures |  Segment number   |     |
           |    in this segment   |  of this segment  |     |
           |------------------------------------------|     |
  |--------|  Procedure #1               PASCALSYSTEM |     |
  |        |------------------------------------------|     | Procedure
  |  |-----|  Procedure #2                CLEARSCREEN |     | Dictionary
  |  |     |------------------------------------------|     |
  |  |  |--|  Procedure #3                            |     |
  |  |  |  |------------------------------------------|     |
  |  |  |  |                rest of                   |     |
  |  |  |  |          procedure  dictionary           |     |
  |  |  |  |                                          |   --|
  |  |  |  |==========================================|
  |  |  |  |                                          |   --|
  |------->|  PROCEDURE #1  (outer block of           |     |
     |  |  |                 PASCALSYSTEM)    code    |     |
     |  |  |------------------------------------------|     |
     |  |  |  Other procedures in             code    |     | Procedure
     |  |  |  segment PASCALSYSTEM            code    |     | Code
     |  |  |------------------------------------------|     | Sections
     |  |->|  PROCEDURE #3                    code    |     |
     |     |------------------------------------------|     |
     |---->|  PROCEDURE #2  (CLEARSCREEN)     code    |     |
           |==========================================|   --|

                      low diskette addresses
```

FIGURE 2   DETAIL OF THE "PASCALSYSTEM" CODE SEGMENT

A PROCEDURE DICTIONARY

Each procedure in a code segment is assigned a procedure number, starting at 1 for the outer block (the main program or a segment procedure), and ranging as high as 149 . All references to a procedure are made via its number. Translation from a procedure's number to the location of that procedure's code in the code segment is accomplished with the procedure dictionary at the end of the segment. This dictionary is an array indexed by the procedure number. Each array element is a self-relative pointer to the code for the corresponding procedure. Since zero is not a valid procedure number, the zero-th entry of the dictionary is used to store the number of the code segment (even byte) and the number of procedures in that code segment (odd byte).

A PROCEDURE CODE SECTION

A more detailed diagram of the code section for a single procedure within a code segment is seen in Figure 3. That figure shows the code section for procedure CLEARSCREEN, in "The Program"'s main-program code segment, PASCALSYSTEM. Each procedure's code section consists of two parts: the procedure code itself (in the lower portion of the section) and a table of attributes of the procedure. These attributes are:

LEX LEVEL: Odd byte. Specifies the depth of absolute lexical nesting for the procedure. (E.g., the Lex Level (LL) of PASCALSYSTEM = -1, LL of COMPILER or CLEARSCREEN = 0 , LL of COMPINIT = 1 , etc.)

PROCEDURE NUMBER: Even byte. Refers to the number given to this procedure in the procedure dictionary of the parent code segment. For example, CLEARSCREEN is Procedure #2 (see Figure 2).

ENTER IC: A self-relative pointer to the first instruction to be executed for this procedure.

EXIT IC: A self-relative pointer to the beginning of the block of procedure instructions which must be executed to terminate procedure properly.

PARAMETER SIZE: The number of bytes of parameters passed to a procedure from its caller.

DATA SIZE: The size of the activation record (see the later sections of this appendix for details) in bytes, excluding the Markstack and PARAMETER SIZE.

Between these attributes and the procedure code there may be an optional section of memory called the "jump table". Its entries are addresses within the procedure code. JTAB is a term commonly applied to the six attributes just discussed and the jump table itself. JTAB is also one of the system registers, which points to the attributes and jump table section of the currently executing procedure.

```
|----------------|
| PASCALSYSTEM's |
| Procedure      |            high diskette addresses
| Dictionary     |
| Pointer        |      |============================|   --|
|----------------|----->|  Lex Level   | Procedure # |     |
                        |----------------------------|     |
                  |-----|          Enter IC          |     |
                  |     |----------------------------|     |
                  |  |--|          Exit IC           |     | Table of
                  |  |  |----------------------------|     | Attributes
                  |  |  |       Parameter Size       |     | (JTAB)
                  |  |  |----------------------------|     |
                  |  |  |         Data Size          |     |
                  |  |  |----------------------------|     |
                  |  |  |   (Optional Jump Table)    |   --|
                  |  |  |============================|
                  |  |->|                            |   --|
                  |     |        CLEARSCREEN         |     | Procedure
                  |     |           Code             |     | Code
                  |---->|                            |     |
                        |============================|   --|

                           low diskette addresses
```

FIGURE 3   DETAIL OF THE "CLEARSCREEN" PROCEDURE CODE SECTION

SYSTEM MEMORY USE

APPLE II MEMORY MAP

Figure 4 is a sketch of the Apple II′s memory, when running under the Apple Pascal operating system.

This memory map is specific to the Apple II, and does not apply to any other computer. It is provided for your curiosity only: a primary task of the transportable Apple Pascal system is to eliminate the necessity for the programmer to know anything about specific memory addresses and use.

The Apple Pascal file SYSTEM.PASCAL roughly corresponds to "The Program"′s resident code segment named PASCALSYSTEM as discussed throughout this appendix.

```
Address (Hex)
                       Apple Language Card              (Language Card)
  64K (FFFF)  |==============================|     |-------------------|
              |   SYSTEM.PASCAL (Part 2)     |<-->|    Monitor ROM     |
              |------------------------------|     |-------------------|
              |         SYSTEM.APPLE         |
  56K (E000)  |- - - - -    P-code    - - - -|     |-------------------|
              |          Interpreter         |<-->| _ _Interpreter_ _  |
              |    (written in 6502 code)    |     | BIOS I/O Routines |
  52K (D000)  |==============================|     |-------------------|

                      Apple II Main Board
  52K (D000)  |==============================|
              |   I/O Expansion ROM Space    |
              |- - - - - - - - - - - - - - - |
              | I/O Device Addresses & ROMs  |
  48K (C000)  |==============================|
              |            SYSCOM            |
              |------------------------------|
              |    SYSTEM.PASCAL (Part 1)    |
  44K (B000)  |    The Operating System      |
              |------------------------------|
              |              |               |
              |    Pascal Program Stack      |
              |       (builds down)          |
              |              |               |
              |              V               |
              | - - - - - - - - - - - - - -  |  <-- KP      (Top of
              |        (Free Memory)         |     A      Program Stack)
              |                              |     |
  24K (6000)  |------------------------------|     |
              |  High-Res Graphics, Page 2   |     |  (Free
  16K (4000)  |------------------------------|     |  Memory)
              |  High-Res Graphics, Page 1   |     |
   8K (2000)  |------------------------------|     |
              |                              |     V
              | - - - - - - - - - - - - - -  |  <-- NP    (Top of Heap)
              |    Pascal Data Heap      A   |
              |      (builds up)         |   |
   3K  (C00)  |------------------------------|
              |        Text Screens          |
              |------------------------------|
              |    Disk & Console Buffers    |
  .5K  (1FF)  |------------------------------|
              |    Evaluation Stack      |   |
              |     (builds down)        V   |
              | - - - - - - - - - - - - - -  |  <-- SP     (Top of
 .25K  (100)  |------------------------------|          Evaluation Stack)
              | Zero Page Pascal System Use  |
   0K  (000)  |==============================|
```

FIGURE 4   MEMORY MAP OF THE APPLE II
WHEN USING APPLE PASCAL

THE PROGRAM STACK

Figure 5 is a snapshot of user memory, showing the Pascal program stack in some detail, during the execution of a call to procedure CLEARSCREEN from "The Program"'s line C, in segment procedure COMPINIT.

```
                         high memory addresses

                 |================================|
                 |             SYSCOM             |  <-- Segment Dictionary
                 |================================|          Information
                 |          PASCALSYSTEM          |
                 |     Resident Code Segment      |
                 |--------------------------------|
                 |          PASCALSYSTEM          |
                 |   Resident Activation Record   |
                 | - - - - - - - - - - - - - - -  |
                 |           Markstack            |
                 |================================|
                 | COMPILER          Code Segment |
                 |--------------------------------|
                 | COMPILER     Activation Record |
                 | - - - - - - - - - - - - - - -  |
                 |           Markstack            |
                 |================================|
                 | COMPINIT          Code Segment |
                 |--------------------------------|
                 | COMPINIT     Activation Record |
                 | - - - - - - - - - - - - - - -  |
                 |           Markstack            |
                 |================================|
                 | CLEARSCREEN Activation Record  |
                 | - - - - - - - - - - - - - - -  |
                 |           Markstack            |
         MP -->  |================================|  <-- KP  (Top of
(Top Markstack)  |                                |       Program Stack)
                 |                                |
                 |        ( Free Memory )         |
                 |                                |
                 |                                |
                 |--------------------------------|  <-- NP (Top of Heap)
                 |            H E A P             |
                 |================================|
                        low memory addresses
```

FIGURE 5 DETAIL OF USER MEMORY DURING EXECUTION OF PROCEDURE "CLEARSCREEN"

The Pascal heap is at the lowest part of memory available to programs; it grows toward high memory. It is used to store dynamic variables, text files used by the Editor, and other data. The system communications area (also called SYSCOM), is at the top of memory, above the system's resident code segment. SYSCOM is accessible both to assembly-language routines in the interpreter and (as if it were part of the stack) and to system routines coded in Pascal. SYSCOM serves as an important communication link between these two levels of the system (for more details about SYSCOM, see this manual's appendix, ARCHITECTURE OF THE P-MACHINE).

The program stack, growing down from high memory, is used to store three types of items:

1. A Code Segment for each active program segment (see Figures 1, 2, and 3) and for each active UNIT.

2. An Activation Record containing local variables and Markstack parameters for each procedure activation (see Figure 6).

3. A Data Segment for each INTRINSIC UNIT which requires one, loaded on the program stack just before the code segment for that UNIT.

When segment procedure COMPINIT is called in line B of "The Program", COMPINIT's code segment (which includes all the compiler initialization procedures) is loaded onto the program stack. The COMPINIT activation record is then built on top of the program stack.

Consider the status of operations in COMPILER, just as COMPINIT is called in line B. The system registers (see this manual's appendix ARCHITECTURE OF THE P-MACHINE) contain the following:

SP: evaluation Stack Pointer. Points to the current top of the evaluation stack.

KP: program stacK Pointer. Points to the current top of the program stack, just beyond COMPILER's activation record.

IPC: Interpreter Program Counter. Points to the next COMPILER instruction, immediately following line B.

SEG: SEGment pointer. Points to the COMPILER code segment's procedure dictionary.

JTAB: Jump TABle pointer. Points to the table of attributes in COMPILER's procedure code section.

MP: Markstack Pointer. Points to the Markstack in COMPILER's activation record. Used to find variables local to COMPILER.

The call to procedure COMPINIT causes the operating conditions which existed in the system registers, just at the time of COMPILER's call to COMPINIT, to be stored in COMPINIT's Markstack in the following manner:

System registers, at COMPINIT call		Stored in these COMPINIT Markstack fields
SP	--->	MSSP (MarkStack Stack Pointer)
IPC	--->	MSIPC (MarkStack Interpreter Program Counter)
SEG	--->	MSSEG (MarkStack SEGment pointer)
JTAB	--->	MSJTAB (MarkStack Jump TABle)
MP	--->	MSDYN (MarkStack DYNamic link)

In addition, the MarkStack STATic link field (MSSTAT) becomes a pointer to the activation record of the lexical parent of the called procedure. In particular, it points to the MSSTAT field of the parent's markstack. In this example, COMPINIT's MSSTAT is made to point to the MSSTAT field of COMPILER's Markstack. After building the new procedure's activation record on the program stack, new values for the system registers SP, IPC, SEG, JTAB, and MP are established for the new procedure.

If the called procedure has a lex level of -1 or 0, the contents of register BASE are saved on the evaluation stack, and a new value for BASE is calculated. Finally, KP is saved on top of the program stack, and a new value for KP is calculated. These elements are not part of the COMPINIT Markstack or activation record.

AN ACTIVATION RECORD

Figure 6 is a diagram of the activation record which is placed on the stack for COMPINIT.

```
                        high memory addresses

            |============================|   --|
            |  Other COMPINIT variables  |     |
            |----------------------------|     |
            |           BOOL             |     | Local
            |----------------------------|     | Variables
            |            I               |     |
            |----------------------------|     |
            |            J               |   --|  Passed
            |============================|   <----Parameters
            |           MSSP             |   --|  (if any)
            | - - - - - - - - - - - - - -|     |
            |           MSIPC            |     |
            | - - - - - - - - - - - - - -|     |
            |           MSSEG            |     |
            | - - - - - - - - - - - - - -|     | Markstack
            |           MSJTAB           |     |
            | - - - - - - - - - - - - - -|     |
            |           MSDYN            |     |
            | - - - - - - - - - - - - - -|     |
            |           MSSTAT           |     |
     MP --> |============================|   --|
```

FIGURE 6   DETAIL OF THE "COMPINIT" ACTIVATION RECORD

In the upper portion of the activation record, space is allocated for variables local to the new procedure. For example, COMPINIT's activation record allocates space for integer variables I and J, as well as boolean variable BOOL.

Note that in this example no space is needed for passed parameters because none were passed to this procedure. If parameters are passed, they occupy space after the last local variable. If the procedure is a function, space is also reserved (following the last passed parameter) for storing the function's returned value.

The lower portion of the activation record is called a "Markstack" (also sometimes called a Mark Stack Control Word, or MSCW). When a call to any procedure is made, the current values of the system registers, which characterize the operating environment of the calling procedure, are stored in the Markstack of the called procedure. This allows the system registers to be restored to pre-call conditions when control is returned to the calling procedure.

When the call to CLEARSCREEN is made in line C of "The Program", another activation record is added to the program stack. Once again the register values and the appropriate Static Link are stored in the new Markstack (in CLEARSCREEN's activation record), and the system registers are then updated. Note that the new SEG no longer points to the COMPINIT segment's procedure dictionary, but to the procedure dictionary for the PASCALSYSTEM code segment (which contains procedure CLEARSCREEN).

No code segment for CLEARSCREEN is added to the program stack before building the activation record, since the code for CLEARSCREEN is already present on the program stack, in the code segment for PASCALSYSTEM. The invocation of CLEARSCREEN causes only an activation record to be added to the program stack. When CLEARSCREEN and COMPINIT are completed, the COMPILER activation record will again be the top element on the stack.

MORE ON THE PROGRAM STACK

Figure 7 is a more detailed diagram of the program stack during execution of an instruction in CLEARSCREEN, including appropriate pointers for Static and Dynamic Links of CLEARSCREEN's Markstack. Note where the system registers point in the program stack. In particular, JTAB points to the table of attributes in the CLEARSCREEN procedure code section which is in the PASCALSYSTEM code segment, IPC points to the next instruction inside that CLEARSCREEN code, and SEG points to the base of the PASCALSYSTEM code segment's procedure dictionary. SP points to the top of the evaluation stack, which is not shown in this diagram.

OVERVIEW

SUMMARY OF THE FIGURES

Figure 8 illustrates a top-down process by showing the relationships among Figures 1 through 7.

```
              high memory addresses
          |=============================|<--- SEG
          | PASCALSYSTEM   Code Segment |
          | (includes CLEARSCREEN code) |<--- JTAB, IPC
          |-----------------------------|
          | PASCALSYSTEM Activation Rec.|
          | - - - - - - - - - - - - - - |
|--------->|          Markstack          |
|         |=============================|
|         | COMPILER       Code Segment |
|         |-----------------------------|
|         | COMPILER  Activation Record |
|         | - - - - - - - - - - - - - - |
|         |          Markstack          |
|         |=============================|           --|
|   |---->|    20        |        4     |             |
|   |     |- - - - - - - - - - - - - - -|             | Code
|   |     |COMPINIT Procedure Dictionary|             | Segment
| |------>|-----------------------------|             |  of
| | | |-->| COMPINIT  outer block code  |             | COMPINIT
| | | |   | - - - - - - - - - - - - - - |             |
| | | |   | COMPINIT  procedures  code  |           --|
| | | |   |=============================|
| | | |   | COMPINIT  local variables   |           --| Activation
| | | |   | - - - - - - - - - - - - - - |             | Record
| | | |   | COMPINIT   Markstack        |<-|        --| of COMPINIT
| | | |   |=============================|  |
| | | |   | CLEARSCREEN local variables |  |        --|
| | | |   |-----------------------------|  |          |
| | | |   |           MSSP              |----------|  |
| | | |   | - - - - - - - - - - - - - - |  |       |  |
| | | |---|           MSIPC             |  |       |  | Activation
| | |     | - - - - - - - - - - - - - - |  |       |  | Record
| | |-----|           MSSEG             |  |       |  |  of
| |       | - - - - - - - - - - - - - - |  |       |  | CLEARSCREEN
| |-------|           MSJTAB            |  |       |  |
|         | - - - - - - - - - - - - - - |  |       |  |
|         |           MSDYN             |--|       |  |
|         | - - - - - - - - - - - - - - |          |  |
|---------|           MSSTAT            |          |--|
          |=============================| <- MP,   |
          |        ( Free Memory )      | & KP     V
          |-----------------------------|        (to
               low memory addresses            evaluation
                                                 stack)
```

FIGURE 7 DETAIL OF THE PROGRAM STACK DURING
EXECUTION OF PROCEDURE CLEARSCREEN

```
  A PROGRAM CODEFILE

|====================|
| Main PASCALSYSTEM  |
|  Program Segment   |-->
|--------------------|
|   PASCALSYSTEM's   |
| Segment Procedures |
|====================|
| Segment Dictionary |
|====================|
```

Figure 1
Complete Codefile
of "The Program"

```
      A CODE SEGMENT

|========================|
|  Procedure Dictionary  |
|========================|
| PASCALSYSTEM     code  |
| outer block   section  |
|------------------------|
| More procedures   code |
| in segment    sections |
| PASCALSYSTEM           |
|------------------------|
| Procedure        code  |-->
| CLEARSCREEN   section  |
|========================|
```

Figure 2
Detail of the
"PASCALSYSTEM"
Code Segment

```
  A PROCEDURE
 CODE SECTION

|==============|
|   Table of   |
|  Attributes  |
|    (JTAB)    |
|--------------|
| CLEARSCREEN  |
|  Procedure   |
|     Code     |
|==============|
```

Figure 3
Detail of the
"CLEARSCREEN"
Procedure
Code Section

```
      MEMORY MAP

|==================|
|        |         |
| The Program Stack|-->
|        |         |
|_ _ _ _ V_ _ _ _ _|
|                  |
| ( Free Memory )  |
|_ _ _ _ _ _ _ _ _ |
|        A         |
|        |         |
|     The Heap     |
|==================|
|__________________|
|Evaluation Stack ||
|_ _ _ _ _ _ _ _ V_|
|==================|
```

Figure 4
Memory Map
of the Apple II

```
     THE PROGRAM STACK

|==========================|
|       PASCALSYSTEM       |
|       Code Segment       |-->
|- - - - - - - - - - - - - |
|       PASCALSYSTEM       |
|    Activation Record     |
|--------------------------|
| COMPILER    Code Segment |
|- - - - - - - - - - - - - |
| COMPILER Activation Rec. |
|--------------------------|
| COMPINIT    Code Segment |
|- - - - - - - - - - - - - |
| COMPINIT Activation Rec. |-->
|--------------------------|
| CLEARSCREEN Activ'n Rec. |
|==========================|
```

Figure 5
Detail of the Program Stack
During Execution of the
"CLEARSCREEN" Procedure

```
 A CODE SEGMENT

|--------------|
| ( Figure 2 ) |
|--------------|

AN ACTIVATION
   RECORD

|=============|
|  COMPINIT   |
|    Local    |
|  Variables  |
|-------------|
| Parameters  |
|-------------|
|  Markstack  |
|=============|
```

Figure 6
Detail of the
"COMPINIT"
Activation Record

Figure 7
The Program Stack in More Detail

FIGURE 8 RELATIONSHIP OF APPENDIX FIGURES

"THE PROGRAM"

This is "The Program", various parts of which are used as examples throughout this appendix. As mentioned in the introduction, "The Program" shows just the partial skeleton of a very early version of the Apple Pascal operating system. Much of the code is only hinted at, and many of the segments used in the current version of the Apple Pascal operating system are missing entirely from "The Program".

```
PROGRAM PASCALSYSTEM;
VAR
  SYSCOM: SYSCOMREC;
  CH:CHAR;

  PROCEDURE CLEARSCREEN; FORWARD;

  SEGMENT PROCEDURE USERPROGRAM;
    BEGIN
     ...
    END;
  SEGMENT PROCEDURE COMPILER;
  VAR
    SY,OP:INTEGER;
    SYMCURSOR:INTEGER;

     PROCEDURE INSYMBOL; FORWARD;

     SEGMENT PROCEDURE COMPINIT;
     VAR
       I,J:INTEGER;
       BOOL:BOOLEAN;
     BEGIN
       ...
       I:=1;
       CLEARSCREEN;    <--------------------------------Line C
       INSYMBOL;
       ...
     END;

     PROCEDURE INSYMBOL;
     BEGIN ... END;

     PROCEDURE BLOCK;
     BEGIN   ...  END;
    BEGIN (*COMPILER*)
     ...
     COMPINIT;         <--------------------------------Line B
     INSYMBOL;
     ...
    END; (*COMPILER*)
```

(continued on next page)

```
   SEGMENT PROCEDURE EDITOR;
     BEGIN   ...   END;

   PROCEDURE CLEARSCREEN
     BEGIN
       ...
       WRITE(-------------------);
       ...
     END;

BEGIN (*PASCALSYSTEM*)
  REPEAT
    READ(CH);
    CASE CH OF
      'C':COMPILER;    <--------------------------------Line A
      'E':EDITOR;
      'U':USERPROGRAM
      ...
    END (*CASE*)
  UNTIL CH = 'H'
END.
```

APPENDIX C

FILE FORMATS

266 TEXT FILES
266 DATA FILES
266 CODE FILES

TEXT FILES

At the beginning of each text file is a 1024-byte (two blocks, on diskette) header page, which contains information for the use of the text editor. This space is reserved for use by the text editor, and is respected by all portions of the system. When a user program opens a TEXT file, and REWRITEs or RESETs it with a title ending in .TEXT , the I/O subsystem will create and skip over the initial header page. This is done to facilitate users editing their input and/or output data. The file-handler will transfer the header page only on a disk-to-disk transfer, and will omit it on a transfer to a serial device (thus transfers to PRINTER:, and CONSOLE: will omit the header page).

Following the initial header page, the text itself appears in subsequent 1024-byte text pages (two blocks each, on diskette), where a text page is defined:

```
    [DLE] [indent] [text] [CR] [DLE] [indent] [text] [CR]...[nulls]
```

DLE's (Data Link Escapes) are followed by an indent-code, which is a byte containing the value 32+(number to indent). The nulls at the end of the page follow a [CR] in all cases, and are a pad to the end of a 1024-byte page (because the compiler wants integral numbers of lines on a page). The Data Link Escape and corresponding indentation code are optional. In a given text file, some lines will have the codes, and some won't.

DATA FILES

The formats for Data files are up to the user.

CODE FILES

Codefiles may contain up to 16 segments. Block 0 of a codefile contains information regarding name, kind, relative address and length of each code segment. This information is called the "segment dictionary" and is represented as a record:

```
        RECORD
          DISKINFO: ARRAY[0..15] OF
                      RECORD
                        CODELENG, CODEADDR:INTEGER
                      END;

          SEGNAME: ARRAY[0..15] OF PACKED ARRAY[0..7] OF CHAR;

          SEGKIND: ARRAY[0..15] OF (LINKED,HOSTSEG,SEGPROC,UNITSEG,
                                    SEPRTSEG,UNLINKED_INTRINS,
                                    LINKED_INTRINS,DATASEG);
```

```
      TEXTADDR: ARRAY[0..15] OF INTEGER;

      SEGINFO: PACKED ARRAY [0..15] OF
                               PACKED RECORD
                                  SEGNUM:  0..255;
                                  MTYPE:  0..15;
                                  UNUSED:  0..1;
                                  VERSION_:  0..7
                               END;

      INTRINS_SEGS: SET OF 0..31;
      (* library information: format undefined *)

      COMMENT: STRING
    END;
```

First is an array of sixteen word-pairs, each word-pair describing one segment of code. CODELENG and CODEADDR give, respectively, the length of the code segment in bytes, and the block address of the code segment.

The first and second word in the first block constitute the first word-pair, which describes the block-location and length of code segment 0. Segment 0 contains the outermost code for the main program. Subsequent segments contain the code for the program's various segment procedures and regular Units (if any), in the order of their appearance in the program.

Following this word-pair array is an array of arrays of characters. This is an array of sixteen eight-character arrays which describe the segments by name. These eight characters are those which identify the main program and its segment procedures at compile time.

Following the array of names is an array, again sixteen words long, of state descriptors. The values in this array indicate what kind of segment (SEGKIND) is at the described location. The values for this array, at present, are: LINKED, HOSTSEG, SEGPROC, UNITSEG, SEPRTSEG, UNLINKED INTRINSIC, LINKED INTRINSIC, and DATASEG. A description of the segments corresponding to various SEGKINDs follows:

LINKED	A fully executable code segment. Either all external references (UNITs or EXTERNALs or .REFs) have been resolved, or none were present.

HOSTSEG	The outer block of a Pascal program, if the program has external references.
SEGPROC	A Pascal segment procedure (not used).
UNITSEG	A compiled regular UNIT .
SEPRTSEG	A separately compiled procedure or function. Assembly-language codefiles are always of this type.
UNLINKED_INTRINS	An INTRINSIC UNIT containing unresolved calls to assembly-language procedures or functions.
LINKED_INTRINS	An INTRINSIC UNIT in its final, ready-to-run state.
DATASEG	A specification for the data segment associated with an INTRINSIC UNIT, telling how many bytes to allocate and which segment to use.

See the Apple Pascal Language Reference Manual for more information about Pascal SEGMENTs and UNITs.

After the array of segment kinds is an array of sixteen integers. If a segment is a regular or Intrinsic Unit, the value of the corresponding array element gives the relative block number (TEXTADDR) where the Interface portion of that Unit begins. Array elements corresponding to non-Unit segments have the value zero.

Next is an array of sixteen words (SEGINFO), each word describing one segment of code. Bits 0 through 7 (the rightmost bits) of each word specify the segment number for that code segment. This specifies the slot number the code segment will occupy in the system's SEGTABLE, at execution time. Bits 8 through 11 identify the "Machine type" which tells what kind of code is present in the code segment. These machine types are assigned as follows:

0	Unidentified code, perhaps from a previous compiler.
1	P-code, most significant byte first (positive byte sex).
2	P-code, least significant byte first (negative byte sex). A stream of packed ACCII characters fills the low byte of a word first, then the high byte. This is the kind of P-code used by the Apple.
3 through 9	Assembled machine code, produced from assembly-language text. Machine type 7 identifies machine code for Apple's 6502.

Bit 12 (UNUSED) in each word is an unused filler, usually set to zero. Bits 13 through 15 identify the version number of the system; currently this is set to the number one.

Next are two words (INTRINS_SEGS) that tell the system which Intrinsic Units are needed in order to execute the codefile. Each Intrinsic Unit in SYSTEM.LIBRARY is identified by a segment number (or two segment numbers, if the Intrinsic Unit has a data segment). Each of the thirty-two bits in INTRINS_SEGS corresponds to one of the thirty-two possible Intrinsic Unit segment numbers. If the n-th bit is set to 1, this indicates the program will USE the Intrinsic Unit whose segment number in SYSTEM.LIBRARY is n .

Library information of undefined format occupies most of the remainder of the segment dictionary block. The "copyright" text supplied by the Pascal Computer $C option may appear at the very end of the block. The actual code segments begin in block 1 of the codefile. The internal format of codefile segments is shown in some detail in the first portion of this manual's appendix, OPERATION OF THE P-MACHINE. See that appendix for more details about codefile segments.

For an unlinked code segment (that is, a segment containing unresolved external references) the Compiler generates Linker information which begins at the first block boundary following the last segment of code. This information is a series of records, one for each UNIT, routine or variable which is referenced in, but not defined in the source. The first eight words of each record contain the following information:

```
LITYPES = (EOFMARK, UNITREF, GLOBREF, PUBLREF, PRIVREF, CONSTREF,
           GLOBDEF, PUBLDEF, CONSTDEF, EXTPROC, EXTFUNC, SEPPROC,
           SEPFUNC, SEPPREF, SEPFREF);
LIENTRY=RECORD
     NAME: ALPHA;
     CASE LITYPE:  LITYPES OF
       UNITREF,
       GLOBREF,
       PUBLREF,
       PRIVREF,
       SEPPREF,
       SEPFREF,
       CONSTREF:
             (FORMAT: OPFORMAT;        (format of lientry.name can be
                                          any of BIG, BYTE or WORD )
              NREFS: INTEGER;          (number of refs to lientry.name
                                          in compiled code segment)
              NWORDS: LCRANGE);        (size of privates in words)
              GLOBDEF:
                  (HOMEPROC: PROCRANGE;  (which procedure it occurs in)
                  ICOFFSET:ICRANGE);     (byte offset in p-code)
```

```
    PUBLDEF:
           (BASEOFFSET: LCRANGE);    (compiler-assigned word offset)
    CONSTDEF:
            (CONSTVAL: INTEGER);     (user's defined value)
    EXTPROC, EXTFUNC,
    SEPPROC, SEPFUNC:
            (SRCPROC: PROCRANGE;     (procedure number in source seg)
             NPARAMS: INTEGER);      (number of parameters expected)
    EOFMARK:
            (NEXTBASELC: LCRANGE)    (private var allocation info)
  END(lientry);
```

If the LITYPE is one of the first case variants, then following this portion of the record is a list of pointers into the code segment. Each of these pointers is the absolute byte address within the code segment of a reference to the variable, UNIT or routine named in the lientry. These are eight-word records, but only the first NREFs of them are valid.

APPENDIX D

TABLES

272 When to Use .TEXT and .CODE
273 Language System Diskette Files
276 Pascal I/O Device Volumes
277 Apple I/O Device Slots
278 Execution Error Messages
280 I/O Error Messages
281 6502 Assembler Error Messages
283 ASCII Character Codes
284 P-Machine Op-Codes

WHEN TO USE .TEXT AND .CODE

An Apple Pascal filename normally ends with a "suffix" that tells the system something about the contents of that file. The most common suffixes are .TEXT , for files containing text (natural language, Pascal program text, or 6502 assembly-language text), and .CODE , for files containing code (compiled P-code or assembled 6502 machine code).

Many Apple Pascal commands deal with various diskette files that you must specify by filename. In those instances where the file being acted on can be of only one type (.TEXT , .CODE , etc.) the system allows you to type the filename either with or without the suffix. If you forget to add the suffix, the system will add it for you. In those instances where a command may apply to files of more than one file type, the following rules apply:

1. Several Filer commands which must distinguish between textfiles and codefiles will use the given filename exactly as typed, without adding any suffix for you. These commands are: T(ransfer, M(ake, C(hange, and R(emove.

2. The S(ave command automatically supplies the correct suffix (.TEXT or .CODE) to the version of the workfile being S(aved. Therefore, when using this command, don't specify a suffix.

3. Code segments may be stored in library files which have either .CODE or .LIBRARY as their suffix. Commands that use these files must specify the suffix. These include the Librarian utility's OUTPUT CODE FILE, and the Pascal Compiler's Use-Library (*$Ufilename*) control option.

LANGUAGE SYSTEM DISKETTE FILES

The following table shows which files are normally found on each of the Language System Diskettes needed for Apple Pascal. The ORDER of the files on any diskette is unimportant. When most files are needed by the system, it is only necessary that the file be present on ANY diskette in ANY drive. For exceptions to this rule, see the DISKFILES NEEDED sections of this manual's chapter THE COMMAND LEVEL.

Diskette
APPLE0:

SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.COMPILER
SYSTEM.EDITOR
SYSTEM.FILER
SYSTEM.LIBRARY
SYSTEM.CHARSET
SYSTEM.SYNTAX

Diskette
APPLE1:

SYSTEM.APPLE
SYSTEM.PASCAL
SYSTEM.MISCINFO
SYSTEM.EDITOR
SYSTEM.FILER
SYSTEM.LIBRARY
SYSTEM.CHARSET
SYSTEM.SYNTAX

Diskette
APPLE2:

SYSTEM.COMPILER
SYSTEM.LINKER
SYSTEM.ASSMBLER
6500.OPCODES
6500.ERRORS

Diskette
APPLE3:

SYSTEM.APPLE
FORMATTER.CODE
FORMATTER.DATA
LIBRARY.CODE
LIBMAP.CODE
SETUP.CODE
BINDER.CODE
CALC.CODE
LINEFEED.TEXT
LINEFEED.CODE
SOROCGOTO.TEXT
SOROCGOTO.CODE
SOROC.MISCINFO
HAZELGOTO.TEXT
HAZELGOTO.CODE
HAZEL.MISCINFO
CROSSREF.TEXT
CROSSREF.CODE
SPIRODEMO.TEXT
SPIRODEMO.CODE
HILBERT.TEXT
HILBERT.CODE
GRAFDEMO.TEXT
GRAFDEMO.CODE
GRAFCHARS.CODE
GRAFCHARS.TEXT
TREE.TEXT
TREE.CODE
BALANCED.TEXT
BALANCED.CODE
DISKIO.TEXT
DISKIO.CODE

The next portion of the table gives more information about the various files provided with the Apple Pascal system.

Filename	Contents of File	Use of File	When File Needed
SYSTEM.APPLE	Interpreter, written in 6502 machine language	Executes P-code on Apple's 6502 processor	Power-on, H(alt
SYSTEM.PASCAL	Command level portion of operating system	Lets you pick E(dit, F(ile, R(un, etc.	Power-on, H(alt, RESET, I(nitialize, every return to Command level
SYSTEM.MISCINFO	Information about terminal in use	Tells system about terminal hardware	Power-on, H(alt, RESET, I(nitialize
SYSTEM.EDITOR	Text Editor	Lets you make & change text	E(dit, C(ompile R(un, A(ssemble
SYSTEM.FILER	Filer	Lets you store, delete & move disk files	F(ile
SYSTEM.LIBRARY	Routines for I/O, long integers, trig. functions, graphics, etc.	Many programs use these library routines	R(un, X(ecute, L(ink, C(ompile, if library routines are used
SYSTEM.CHARSET	Array providing upper & lower case graphic character set	Lets you put text on graphics screen	Used by WCHAR & WSTRING in TURTLEGRAPHICS
SYSTEM.SYNTAX	Compiler error messages	Provides message in E(ditor after Compiler finds an error	R(un, C(ompile followed by E(dit after an error
SYSTEM.COMPILER	Pascal Compiler	Converts Pascal program text to P-code	C(ompile, R(un
SYSTEM.LINKER	Linker	Puts library routines into your program	L(ink, R(un

SYSTEM.ASSMBLER	6502 Assembler	Converts 6502 assembly text into machine code	A(ssemble
6500.OPCODES	Instruction set for Assembler	Used by the Assembler	A(ssemble
6500.ERRORS	Assembler error messages	Provides message after Assembler finds an error	A(ssemble
FORMATTER.CODE FORMATTER.DATA	Utility program	Formats new diskettes	X(ecute FORMATTER
LIBRARY.CODE	Utility program	Puts new routines into library	X(ecute LIBRARY
LIBMAP.CODE	Utility program	Reveals contents of library file	X(ecute LIBMAP
SETUP.CODE	Utility program	Makes new file SYSTEM.MISCINFO for use with external terminal	X(ecute SETUP
SOROC.MISCINFO	SYSTEM.MISCINFO for Soroc IQ120 terminal	Eliminates SETUP for using a Soroc IQ120	
HAZEL.MISCINFO	SYSTEM.MISCINFO for Hazeltine 1500 terminal	Eliminates SETUP for using a Hazeltine 1500	
BINDER.CODE	Utility program	Makes new file SYSTEM.PASCAL with GOTOXY for external terminal	X(ecute BINDER
SOROCGOTO.TEXT SOROCGOTO.CODE	GOTOXY procedure for Soroc IQ120	Used with BINDER for Soroc IQ120	
HAZELGOTO.TEXT HAZELGOTO.CODE	GOTOXY procedure for Hazeltine 1500	Used with BINDER for Hazeltine 1500	
CALC.CODE	Utility program	Lets you divide, multiply, add & subtract numbers	X(ecute CALC
LINEFEED.TEXT LINEFEED.CODE	Utility program	Removes linefeed normally sent to printer after RETURN	X(ecute LINEFEED

CROSSREF.TEXT
CROSSREF.CODE
SPIRODEMO.TEXT
SPIRODEMO.CODE
HILBERT.TEXT
HILBERT.CODE
GRAFDEMO.TEXT
GRAFDEMO.CODE
GRAFCHARS.CODE
GRAFCHARS.TEXT
TREE.TEXT
TREE.CODE
BALANCED.TEXT
BALANCED.CODE
DISKIO.TEXT
DISKIO.CODE

These are a collection of small demonstration programs, illustrating various features of the Apple Pascal language. The first 5 pairs of files show examples of using the Apple Pascal graphics features. The next 2 pairs of files show a simple "tree" algorithm for sorting data. The last pair of files is a brief example of using random-access disk files.

Each pair of files consists of a .TEXT version which you can read in the Editor, and a .CODE version you can X(ecute. See the Apple Pascal Language Reference Manual for more information about these files.

PASCAL I/O DEVICE VOLUMES

The Apple Pascal operating system assigns volume numbers and volume names to the various input/output devices as follows:

Volume Number	Volume Name	Description of Input/Output Device
#0:		(not used)
#1:	CONSOLE:	Screen & keyboard (echo on input)
#2:	SYSTERM:	Screen & keyboard (no echo on input)
#3:		(not used)
#4:	<diskette name>:	Boot disk drive (slot 6, drive 1)
#5:	<diskette name>:	2nd disk drive (slot 6, drive 2)
#6:	PRINTER:	Printer (card in slot 1)
#7:	REMIN:	Remote input (card in slot 2)
#8:	REMOUT:	Remote output
#9:	<diskette name>:	5th disk drive (slot 4, drive 1)
#10:	<diskette name>:	6th disk drive (slot 4, drive 2)
#11:	<diskette name>:	3rd disk drive (slot 5, drive 1)
#12:	<diskette name>:	4th disk drive (slot 5, drive 2)

APPLE I/O DEVICE SLOTS

In using an Apple computer with the Apple Pascal operating system, the Apple's peripheral equipment slots are assigned as follows:

Apple Slot	Input/Output Device and Card Assigned to That Slot	Apple Pascal Operating System Use
0	Apple Language Card *	Stores interpreter and I/O system
1	Printer (Communications Interface Card, Serial Interface Card, or Parallel Printer Interface Card)	PRINTER: or #6:
2	Modem, Apple-to-Apple communication, etc. (Communications Interface Card or Serial Interface Card)	REMIN: or #7: REMOUT: or #8:
3	External terminal (Communications Interface Card. Use of Serial Interface Card is tolerated.)	CONSOLE: or #1: (with echo on input) SYSTERM: or #2: (without echo on input)
4	5th disk drive (Drive 1) 6th disk drive (Drive 2) (Disk Controller Card)	diskette name: or #9: diskette name: or #10:
5	3rd disk drive (Drive 1) 4th disk drive (Drive 2) (Disk Controller Card)	diskette name: or #11: diskette name: or #12:
6	Boot disk drive (Drive 1) * 2nd disk drive (Drive 2) (Disk Controller Card)	diskette name: or #4: diskette name: or #5:
7	Must not contain a disk drive (Do not install a Disk Controller Card here.)	
	Apple keyboard and screen (only if there is no Communications Interface Card in slot #3)	CONSOLE: or #1: (with echo on input) SYSTERM: or #2: (without echo on input)

Note: An asterisk (*) indicates a device which is REQUIRED to be present in that slot.

Note: It is possible to use the Apple Pascal operating system with an Apple computer system containing only a single disk drive.

Note: If a Communications Interface Card is installed in slot #3, for use with an external terminal, booting the system automatically loads the Apple Pascal operating system into a different area of memory (using some of the Apple's text-screen memory). This happens whether or not the external terminal is actually connected to the Communications Interface Card. To return to using the Apple without the external terminal, you must turn off the Apple's power switch, remove the Communications Interface Card from slot #3, and turn on the Apple's power again to re-boot the system.

EXECUTION ERROR MESSAGES

When a program or any portion of the Apple Pascal operating system is running, execution errors are reported by number or by message, in one of the following forms:

```
EXEC ERR # 10
S# 1, P# 7, I# 56
TYPE <SPACE> TO CONTINUE
```

```
IO ERROR: VOL NOT FOUND
S# 1, P# 7, I# 56
TYPE <SPACE> TO CONTINUE
```

where S# specifies the program's current segment number, P# specifies the procedure number within that segment, and I# specifies the byte number in that procedure where the error was detected. User I/O errors (only) are reported in the more detailed second form only if file SYSTEM.PASCAL is in the boot drive at the time.

See this manual's section on Error Handling, in the appendix ARCHITECTURE OF THE P-MACHINE, for more details. Also see the Apple Pascal Language Reference Manual's discussion of the L+ (compiled listing) compiler option, which describes how to list segment, procedure, and byte number information when you compile a program.

Error Number	Error Message and Description	Fatal Error?
0	System error of undefined nature.	FATAL
1	Invalid index, value out of range for string or subrange (XINVNDX). Does not occur if R- compiler option used.	
2	No segment: bad code file (XNOPROC). File reads correctly from disk, but not a valid segment.	
3	Procedure not present at exit time (XNOEXIT): exit from a procedure that was not previously called or active.	

4	Stack overflow (XSTKOVR): the program stack and the heap together have exceeded available user memory.	
5	Integer overflow (XINTOVR). Integer arithmetic gave a result >16 bits. Long integer arithmetic gave an intermediate result >36 digits or final result was assigned to variable of insufficient size.	
6	Divide by zero (XDIVZER).	
7	Invalid memory reference <bus timed out> (XBADMEM): (not used on the Apple).	
8	User break (XUBREAK): "break" key pressed (CTRL-@, on the Apple).	
9	System I/O error (XSYIOER): error in attempting to read an operating system segment from disk.	FATAL
10	User I/O error (XUIOERR): error when user's program attempted a blockread, blockwrite, get, or put. If file SYSTEM.PASCAL available, this error is further reported as an I/O ERROR (see next page).	
11	Unimplemented instruction (XNOTIMP): op-code not implemented, or CSP to non-existent routine.	
12	Floating point math error (XFPIERR): error in real number format, overflow, underflow, etc.	
13	String too long (XS2LONG): attempt to store a source string into a destination string of insufficient size.	
14	Halt, Breakpoint (without debugger in core) (XHLTBPT): (not used on the Apple).	
15	Bad Block (not used on the Apple; Apple reports I/O ERROR #64, instead).	

A fatal error either causes the system to "cold boot" itself or (if the error was totally lethal to the system) forces you to "cold boot" the system by turning the Apple off and then on again. All other errors cause the system to re-initialize itself (do a "warm boot" by calling system procedure INITIALIZE), usually after you press the Apple's spacebar to continue.

I/O ERROR MESSAGES

Error Number	Error Message and Description
0	No error
1	Diskette has bad block: parity error (CRC). (Not used on the Apple.)
2	Bad device (volume) number.
3	Bad mode: illegal operation. (For example, an attempt to read from PRINTER:.)
4	Undefined hardware error. (Not used on the Apple.)
5	Lost device: device is no longer on-line, after successfully starting an operation using that device.
6	Lost file: file is no longer in the diskette directory, after successfully starting an operation using that file.
7	Bad title: illegal filename. (For example, filename is more than 15 characters long.)
8	No room: insufficient space on the specified diskette. (Files must be stored in contiguous diskette blocks.)
9	No device: the specified volume is not on-line.
10	No file: the specified file is not in the directory of the specified volume.
11	Duplicate file: attempt to re-write a file when a file of that name already exists.
12	Not closed: attempt to open an already-open file.
13	Not open: attempt to access a closed file.
14	Bad format: error in reading real or integer. (For example, your program expects an integer input but you typed a character.)
15	Ring buffer overflow: characters are arriving at the Apple faster than the input buffer can accept them.

16	Write-protect error: the specified diskette is write-protected.
64	Device error: failed to complete a read or write correctly (bad address or data field on diskette).

The appropriate one of these I/O error messages is given when execution error #10 occurs (see previous page), if the file SYSTEM.PASCAL is in the boot drive. See the Apple Pascal Language Reference Manual for information about the Apple Pascal function IORESULT, which returns the error numbers shown above.

6502 ASSEMBLER ERROR MESSAGES

When the 6502 Assembler discovers an error in your assembly-language routine, it gives an error message taken from the file 6500.ERRORS , usually found on diskette APPLE2: . If the file 6500.ERRORS is not available in any drive, errors will be reported by number, only.

The 6502 Assembler error message corresponding to each error number is given in the table below. Some people may prefer to gain some additional diskette space by removing the file 6500.ERRORS and using this table instead.

The first portion of this table lists all the general error messages. Machine errors specific to Apple's 6502 are found in the last portion of the table.

GENERAL ERRORS

1: Undefined label
2: Operand out of range
3: Must have procedure name
4: Number of parameters expected
5: Extra garbage on line
6: Input line over 80 characters
7: Not enough .IF's
8: Must be declared in .ASECT before used
9: Identifier previously declared
10: Improper format
11: .EQU expected
12: Must .EQU before use if not to a label
13: Macro identifier expected
14: Word addressed machine
15: Backward .ORG currently not allowed
16: Identifier expected
17: Constant expected
18: Invalid structure

19: Extra special symbol
20: Branch too far
21: Variable not PC relative
22: Illegal macro parameter index
23: Not enough macro parameters
24: Operand not absolute
25: Illegal use of special symbols
26: Ill-formed expression
27: Not enough operands
28: Cannot handle this relative expression
29: Constant overflow
30: Illegal decimal constant
31: Illegal octal constant
32: Illegal binary constant
33: Invalid key word
34: Macro stack overflow: 5 nested limit
35: Include files may not be nested
36: Unexpected end of input
37: This is a bad place for an .INCLUDE file
38: Only labels & comments may occupy column 1
39: Expected local label
40: Local label stack overflow
41: String constant must be on one line
42: String constant exceeds 80 characters
43: Illegal use of macro parameter
44: No local labels in .ASECT
45: Expected key word
46: String expected
47: Bad block, parity error (CRC)
48: Bad unit number
49: Bad mode, illegal operation
50: Undefined hardware error
51: Lost unit, unit is no longer on-line
52: Lost file, file is no longer in directory
53: Bad title, illegal file name
54: No room, insufficient space on disk
55: No unit, no such volume on-line
56: No file, no such file on volume
57: Duplicate file
58: Not closed, attempt to open an open file
59: Not open, attempt to access a closed file
60: Bad format, error in reading real or integer
61: Nested macro definitions illegal
62: '=' or '<>' expected
63: May not .EQU to undefined labels
64: Must declare .ABSOLUTE before 1st .PROC

76: Index register required
77: ′X′ or ′Y′ expected
78: Zero-page address required
79: Illegal use of register
80: Index register expected
81: Ill-formed operand
82: ′X′ expected for indexed addressing
83: Must use ′X′ index register

ASCII CHARACTER CODES

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	′	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	89	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

P-MACHINE OP-CODES

Dec	Hex	Mnemonic	Dec	Hex	Mnemonic	Dec	Hex	Mnemonic
0	00	SLDC 0						
1	01	SLDC 1						
.	.	. .						
.	.	. .						
126	7E	SLDC 126						
127	7F	SLDC 127	170	AA	SAS	213	D5	BPT
128	80	ABI	171	AB	SRO	214	D6	XIT
129	81	ABR	172	AC	XJP	215	D7	NOP
130	82	ADI	173	AD	RNP	216	D8	SLDL 1
131	83	ADR	174	AE	CIP	217	D9	SLDL 2
132	84	LAND	175	AF	EQU	218	DA	SLDL 3
133	85	DIF	176	B0	GEQ	219	DB	SLDL 4
134	86	DVI	177	B1	GRT	220	DC	SLDL 5
135	87	DVR	178	B2	LDA	221	DD	SLDL 6
136	88	CHK	179	B3	LDC	222	DE	SLDL 7
137	89	FLO	180	B4	LEQ	223	DF	SLDL 8
138	8A	FLT	181	B5	LES	224	E0	SLDL 9
139	8B	INN	182	B6	LOD	225	E1	SLDL 10
140	8C	INT	183	B7	NEQ	226	E2	SLDL 11
141	8D	LOR	184	B8	STR	227	E3	SLDL 12
142	8E	MODI	185	B9	UJP	228	E4	SLDL 13
143	8F	MPI	186	BA	LDP	229	E5	SLDL 14
144	90	MPR	187	BB	STP	230	E6	SLDL 15
145	91	NGI	188	BC	LDM	231	E7	SLDL 16
146	92	NGR	189	BD	STM	232	E8	SLDO 1
147	93	LNOT	190	BE	LDB	233	E9	SLDO 2
148	94	SRS	191	BF	STB	234	EA	SLDO 3
149	95	SBI	192	C0	IXP	235	EB	SLDO 4
150	96	SBR	193	C1	RBP	236	EC	SLDO 5
151	97	SGS	194	C2	CBP	237	ED	SLDO 6
152	98	SQI	195	C3	EQUI	238	EE	SLDO 7
153	99	SQR	196	C4	GEQI	239	EF	SLDO 8
154	9A	STO	197	C5	GRTI	240	F0	SLDO 9
155	9B	IXS	198	C6	LLA	241	F1	SLDO 10
156	9C	UNI	199	C7	LDCI	242	F2	SLDO 11
157	9D	LDE	200	C8	LEQI	243	F3	SLDO 12
158	9E	CSP	201	C9	LESI	244	F4	SLDO 13
159	9F	LDCN	202	CA	LDL	245	F5	SLDO 14
160	A0	ADJ	203	CB	NEQI	246	F6	SLDO 15
161	A1	FJP	204	CC	STL	247	F7	SLDO 16
162	A2	INC	205	CD	CXP	248	F8	SIND 0
163	A3	IND	206	CE	CLP	249	F9	SIND 1
164	A4	IXA	207	CF	CGP	250	FA	SIND 2
165	A5	LAO	208	D0	LPA	251	FB	SIND 3
166	A6	LSA	209	D1	STE	252	FC	SIND 4
167	A7	LAE	210	D2		253	FD	SIND 5
168	A8	MOV	211	D3	EFJ	254	FE	SIND 6
169	A9	LDO	212	D4	NFJ	255	FF	SIND 7

APPENDIX E

OPERATING SYSTEM SUMMARY

286 OPERATING SYSTEM
286 COMMAND LEVEL
287 FILER
288 EDITOR
289 COMPILER
289 ASSEMBLER
289 LINKER
290 UTILITIES

OPERATING SYSTEM

The following commands are available at all levels of the system:

CTRL-A	Shows the other 40-character "page" of screen display.
CTRL-Z	Causes the display to follow the cursor right and left.
CTRL-@	"Break" signal; does a "warm boot".
CTRL-F	Stops program output to the screen or printer until the next CTRL-F, without stopping the program.
CTRL-S	Temporarily stops any program or process. On the next CTRL-S, the program continues.
RESET	Does a "warm boot".
Power off-on	Does a "cold boot".

COMMAND LEVEL

1. The Command level is reached automatically, each time the system is booted, RESET, or initialized. It is also reached when any program, including any part of the operating system, is terminated.

2. Use the Command level options to select any of the main subdivisions of the Apple Pascal operating system.

These are the Command level options:

F(ile	Deals with the disks and disk files.
E(dit	Helps you create and change text files.
C(ompile	Converts Pascal program text into executable P-code.
A(ssemble	Converts 6502 assembly text into 6502 machine code.
L(ink	Combines external routines into a Pascal program.
X(ecute	Loads and runs a utility program or other P-code file.
R(un	Executes the workfile, automatically compiling and linking first, if necessary.
D(ebug	Not implemented; do not use this option.
U(ser-restart	Re-executes the last program or option executed.
I(nitialize	Does a "warm boot", like pressing RESET.
H(alt	Does a "code boot", like turning the power on.

FILER

1. From Command level, select F(iler. When FILER prompt line appears, you may remove your boot diskette, if necessary.

2. Use Filer commands to move, save, and erase the workfile and other disk files.

These are the Filer commands:

T(ransfer	Copies a file or entire diskette to another diskette or device. Source diskette must be in a drive to begin.
M(ake	Creates a dummy file on diskette.
C(hange	Renames a file or diskette.
R(emove	Erases a file from its diskette directory.
K(runch	Packs all files together on a diskette.
Z(ero	Erases a directory and renames the diskette.
G(et	Designates a file to be used as the next workfile.
S(ave	Saves the workfile on diskette.
N(ew	Clears the workfile.
W(hat	Tells the original name of the current workfile.
V(olumes	Shows which devices and diskettes are in the system.
L(ist-dir	Shows what files are on a diskette.
E(xt-dir	Shows what files are on a diskette, giving more information.
B(ad-blks	Tests diskette information for correct recording.
X(amine	Tries to fix information reported bad by B(ad-blks.
P(refix	Sets the default volume name.
D(ate	Sets the current date.
Q(uit	Leaves the Filer and returns to Command level. Be sure your boot diskette is in the boot drive.

EDITOR

1. From Command level, select the F(iler. Start a N(ew file or G(et an old file for re-editing (one-drive note: first T(ransfer the old file onto your boot diskette). Q(uit the Filer.

2. From Command level, select the E(ditor. Press the RETURN key if you are beginning a new file. Use the Editor commands to I(nsert, D(elete, X(change, and move text. When you are through, Q(uit and U(pdate the workfile.

3. If this is a program, you may R(un it now. Repeat steps 2 and 3 until the program runs correctly.

4. From the Command level, select the F(iler. S(ave the workfile.

These are the Editor commands:

J(ump	Moves cursor to file's B(eginning, E(nd, or preset M(arker.
P(age	Moves cursor one page.
F(ind /x/	Moves cursor to next "x".

I(nsert	Inserts typed text at cursor.
D(elete	Moving cursor erases text.
Z(ap	Erases all text from cursor to start of last F(ind, R(eplace, or I(nsert.
C(opy	Inserts B(uffer (last insertion or deletion) or diskette F(ile at cursor.
X(change	Replaces character at cursor by typed character.
R(eplace /x//y/	Replaces next "x" by "y".

A(djust	Moves line at cursor right and left.
M(argin	Formats all text between two blank lines (one paragraph).

S(et	Places a M(arker at cursor, or sets E(nvironment options for A(uto-indent, F(illing, margins, etc.
V(erify	Redisplays screen with errors gone.
Q(uit	Leaves the Editor. You may U(pdate the workfile, E(xit without updating, W(rite to any diskette file before returning to Command level or S(ave to your original file.

COMPILER

1. From Command level, select R(un or C(ompile.
2. If a text workfile exists, that file is compiled automatically. Otherwise, you are prompted to specify a source textfile and then to specify a destination codefile.
3. If the Compiler finds an error, select the E(ditor to fix it.

ASSEMBLER

1. From Command level, select A(ssemble.
2. If a text workfile exists, that file is assembled automatically. Otherwise, you are prompted to specify a source textfile and then to specify a destination codefile.
3. Finally, you are prompted to specify an output textfile for the assembly listing, if you want one.
4. If the Assembler finds an error, select the E(ditor to fix it.

LINKER

1. From Command level, select R(un or L(ink.
2. R(un links the compiled workfile to UNITs and routines found in SYSTEM.LIBRARY, automatically. L(ink prompts you to specify a host codefile and then to specify as many library codefiles as needed. Press the RETURN key to stop giving library files.
3. Next, you are prompted to specify a map textfile for storing Linker information. Normally, just press the RETURN key to go on.
4. Finally, you are prompted to specify an output codefile for the linked program.

UTILITIES

1. From Command level, select the X(ecute option. When you are prompted EXECUTE WHAT FILE?, type the name of a utility program.

2. The utility programs enable you to format new diskettes, put routines into a library file, change the system to use an external terminal, change printer output, and use your Apple as a calculator.

These are the Utility programs:

FORMATTER	Prepares new diskettes for use by Apple Pascal.
LIBRARY	Puts UNITs and routines into a library file.
LIBMAP	Shows the contents of a library file.
SETUP	Changes the system for use with an external terminal.
BINDER	Changes the system for use with an external terminal.
LINEFEED	Stops the sending of linefeed after RETURN to printers.
CALC	Lets you add, subtract, divide, and multiply numbers.

INDEX

A

.ABSOLUTE , Assembler
  directive 161, 173
activation record 258-260
address field 25
A(djust ,
  Editor command 113 114, 126
APPLESTUFF , Intrinsic Unit 176
APPLE0, contents of 273
APPLE1, contents of 273
APPLE2, contents of 273
APPLE3, contents of 273
architecture, P-machine 224-245
ASCII character code 283
.ASCII , Assembler
  directive 159, 173
A(ssemble ,
  outer level command 17, 138
Assembler, 136-174
  directives 157-172
  directives, summary 172-174
  diskfiles needed 13, 136-137
  error messages 281-282
  example 142-150
  introduction 136-138
  linkage to assembly
    routines 155-156
  summary 289
  syntax of files 151
  syntax of statements 152-155
  use 138-150
assembly file specification
  syntax 152
asterisk
  and P(refix command 65
  as specification for
    SYSTEM LIBRARY 193
  as volume name
    of system disk 28
  in assembly listings 148
  in file size
    specification 30, 44, 137, 139
A(uto-indent, in Editor Environment
  text formatting
    option 98-101, 119
  and Editor M(argin command 114

B

B(ad Blocks ,
  Filer command 61-62, 69
BINDER , utility 210-213
blocks 25
.BLOCK , Assembler
  directive 160, 173
boot diskette 14, 56
boot disk drive 14
booting the system 7-8, 12
brackets, as file size specifiers 44
.BYTE , Assembler directive 160, 173

C

CALC , utility 216-218, 221
  diskfiles needed 216
  use 217-218
  summary 221
calculator 216-218, 221
C(hange , Filer command 45-47, 68
Changing GOTOXY
  diskfiles needed 211
  example 211-213
  summary 220
code files,
  format of 248-253, 262, 266-270
.CODE (versus .TEXT),
  when to use 272
cold boot 7, 12
colon, and P(refix option 65
  as volume specifier 26, 28
comma, as file separator 33, 45
C(ommand character,
  in Editor 115, 120
Command level 5-20
  Command level options 16-19
  Command options summary 20
  commands usable
    at all levels 9-10
  diskfiles needed 11-16
  overall summary 286
  the operating system 6-9
commands, Command level
  description 16-19
  summary 20, 286
commands, Editor
  description 89-123
  summary 124-125, 288

commands, Filer
  description 34-66
  summary 67-69, 287
Communications
  Interface Card 6, 202
C(ompile , outer level
  command 16, 130
Compiler, 127-132
  diskfiles needed 13, 128-130
  introduction 128
  summmary 289
  use 130-132
CONSOLE: , volume name 26, 56
console, external
  hardware requirements, 202-209
  in SETUP utility 199-202
.CONST , Assembler
  directive 166, 173
copy buffer, in Editor 105-107
C(opy , Editor command 104-107, 126
copying a diskette 42-43
  F(rom a diskette file 104-105
  from the copy B(uffer 105-107
copyright, and library files 192-193
CTRL-A , to see other half
  of text screen 9, 67, 75
CTRL-C , in Editor 78, 87, 88
CTRL-F , to flush output 10, 67
CTRL-I , TAB key 92, 125
CTRL-K , to make [67, 85, 125
CTRL-L , to move cursor
  down 85, 86, 125
CTRL-O , to move cursor up 85, 124
CTRL-Q , to move cursor
  to left margin 125
CTRL-S , to stop execution
  temporarily 10, 67
CTRL-X , to erase line 7, 125
CTRL-Z , to auto-follow
  cursor 10, 67, 75
CTRL-@ , to warm boot 10
cursor, in Editor 85-86, 90-92, 125

D

data field 25
data files, format of 266
D(ate , Filer command 65-66, 69
D(ebug , outer level
  prompt only 19, 20
DEC VT52, external terminal 202
.DEF , Assembler
  directive 168, 173
D(elete ,
  Editor command 88, 101-103, 126
delimiters, in Editor
  for paragraphs 120
  with F(ind 95
  with R(eplace 110
directives, Assembler
  description 157-172
  summary 172-174
directory of diskette 25, 56-60
diskette, names 273
diskette directory 25, 56-60
diskettes, contents of 273
diskettes, damaged 61-64
diskette files needed for
  Assembler 13, 136-138
  Calculator 216
  Changing GOTOXY
    Communication 211
  Command level 11-13
  Compiler 13, 128-130
  Editor 13, 72
  Filer 13, 24
  Formatter 184-185
  Library mapping 194-195
  Linker 13, 176-178
  Removing linefeed from return 214
  System Librarian 187-188
  System reconfiguration 199-200
  system, summary 13-14
dollar sign, as destination
  file specification
  in Filer 36, 67
  in Assembler 138
  in Compiler 131
  in Linker 181
duplicating a diskette 42-43

E

E(dit, outer level command 16, 83
Editor 70-126
  brief scenario 77-83
  command summary 125-126
  commands 89-123
  diskfiles needed 13, 72
  introduction 72
  overall summary 290

.ELSE , Assembler directive 165, 173
.END , Assembler directive 159, 172
.ENDC , Assembler directive 165, 173
.ENDM , Assembler directive 163, 173
Environment options,
  in Editor 118-121, 126
equals sign,
  cursor move
    in Editor 92, 103, 125
  wildcard in Filer 30-33
  wildcard in LIBRARY utility 191
.EQU , Assembler directive 161, 173
error handling by P-machine 226
error messages,
  assembly errors 281-282
  execution errors 278-279
  input/output errors 280
<ESC>, to escape
  from Editor command 87
<ETX>, to accept Editor command 87
evaluation stack, P-machine 229
eXchange , Editor command 107, 126
execution error messages 278-279
E(xit , Editor command 122-123, 126
E(xtended directory list,
  Filer command 59-60, 69
EXTERNAL routines 155, 176-180
external terminal 6, 199-213, 219-220

F

file types, list of 28
F(ile, outer level command 16, 33
file-specification syntax 29, 67
filenames 30, 67
Filer 22-69
  command summary 67
  commands 34-66
  diskfiles needed 13, 24
  introduction 24-25
  overall summary 287
  use 33
files 25, 28-33
  diskette file types 28
  filenames 30, 67
  file size specification 30
  maximum number 25
  maximum size 98
  shorthand filename 30
  wildcards 30-31
  workfiles 29
filling,
  text formatting command 98-101, 125
  and M(argin command 114
F(ind , Editor command 93-97, 126
  L(iteral or T(oken search 94
  Repeat-factor 94
  Same-string option 95
  Set direction 94
  Target string and delimiters 95
FORMATTER , utility 184-186, 218
  diskfiles needed 184-185
  use 185-186
  summary 218
formatting commands in Editor 98-101
formatting diskettes 184-186, 218
.FUNC , Assembler directive 159, 172
function calls, P-machine 240-242

G

G(et , Filer command 51-52, 68
GOTOXY procedure 210-213, 220
greater than character,
  in Editor 91-92, 125

H

H(alt , outer level command 19, 20
  diskfiles needed 14
HAZELGOTO.CODE 210, 211
HAZEL.MISCINFO 199
Hazeltine 1500 6, 202, 210

I

.IF , Assembler directive 165, 173
include-file 132, 140
.INCLUDE , Assembler
  directive 172, 174
I(nitialize,
  outer level command 19, 20
  diskfiles needed 14
initializing a diskette 184-186, 218
input/output devices
  by slot number 277-278
  by volume number 26, 267

input/output error messages 280-281
I(nsert ,
  Editor command 87-88, 97-101, 126
instruction set, P-machine 229-245
.INTERP ,
  Assembler directive 162, 173
interpreter, P-code 7, 128
INTRINSIC UNITS 176-180
I/O error messages 280-281

J

J(ump , Editor command 92-93, 125

K

K(runch , Filer command 49-50, 68

L

L(eft margin,
  Editor environment option 120
less-than character,
  in Editor 91-92, 125
LIBMAP , utility 194-198, 219
  diskfiles needed 194-195
  use 195-196
  summary 219
LIBRARY , utility 186-193, 218-219
  diskfiles needed 187-188
  example 188-193
  use 193
  summary 218-219
library files 188-195
library mapping 194-198, 219
linefeed, removing from
  carriage return 214-215, 220
LINEFEED , utility 214-215, 220
  diskfiles needed 214
  example 197-198
  summary 220
  use 215
L(ink, outer level command 17, 178
Linker, 175-181
  diskfiles needed 13, 176-178
  introduction 176
  summary 289
  use 178-181
.LIST , Assembler Directive 170, 174
L(ist directory ,
  Filer command 56-59, 69
L(iteral or T(oken search, in Editor
  using F(ind 94
  using R(eplace 109

M

.MACRO , Assembler directive 163, 173
.MACROLIST ,
  Assembler directive 170, 174
M(ake , Filer command 44-45, 68
mapfile 181, 195
M(argin , Editor command 114-116, 126
markers, in Editor
  J(ump command 92-93
  S(et command 116-117
memory map, Apple II, 254-255

N

N(ew , Filer command 55, 68
.NOLIST ,
  Assembler directive 170, 174
.NOMACROLIST ,
  Assembler directive 170, 174
.NOPATCHLIST ,
  Assembler directive 170, 174

O

op codes, P-machine, 284
operand formats, P-machine 227-228
operating system 2-3, 6
  command summary 21
  overall summary 285-290
.ORG , Assembler directive 161, 173

P

P-code 16, 128
P-machine 128, 223-264
  instruction set 229-245
  op-codes 284
  operation of 247-264
  technical information 224-228
.PAGE , Assembler directive 171, 174
P(age , Editor command 85, 93, 125
paragraph,
  as defined by Editor 114-115
P(aragraph margin,
  Editor Environment option 120
.PATCHLIST ,
  Assembler directive 170, 174
"pound" sign,
  to indicate block structure 56
Power Down-and-Up 10
P(refix , Filer command 28, 65, 69
PRINTER: , volume name 26
.PRIVATE ,
  Assembler directive 167, 173
procedure dictionary, 251
.PROC , Assembler directive 158, 172
program stack,
  P-machine 229, 256-257
prompt line, command level 6
  Editor 76
  Filer 33
Pseudo-machine (see P-machine)
.PUBLIC ,
  Assembler directive 166, 173

Q

question mark,
  at end of command prompt line 7
  wildcard in Filer 30-33
  wildcard in LIBRARY utility 191
Q(uit , Editor command 88, 122, 126
  Filer command 66, 69

R

reference symbol table,
  in Assembler 142
.REF , Assembler directive 169, 173
registers 224-225
REMIN: , volume name 26
REMOUT: , volume name 26
R(emove , Filer command 48-49, 68
removing linefeed from
  carriage return 214-215, 220
repeat-factor, in Editor 91, 125
  with D(elete 101
  with F(ind 94
  with R(eplace 109
  with Z(ap 103
R(eplace ,
  Editor command 108-113, 126
  Literal or Token search 109
  Repeat factor 109
  Same-string option 111
  Set direction 108
  V(erify option 109
RESET 10
R(eturn , Editor command 123, 126
R(ight margin,
  Editor Environment option 120
ROOT VOLUME 28, 56
R(un ,
  outer level command 18-19, 20, 78
  diskfiles needed 24

S

S(ame-string, Editor option
  with F(ind 95
  with R(eplace 111
S(ave , Editor command 124, 126
S(ave , Filer command 53-55, 68
screen display 6, 75, 91
sectors, on diskettes 25
segment dictionary 250
segments, in codefile 248, 250-251
set direction 91-92, 125
  with F(ind 94
  with R(eplace 108
S(et , Editor command
  E(nvironment 118-121, 126
  M(arker 116-117, 126
SETUP parameters, list of 203-209
SETUP , utility 199-202, 219
  diskfiles needed 199
  external terminal
    requirements 202-208
  summary 219
  use 200

SHIFT M , to make] 67, 85, 125
6500.ERRORS 136
6500.OPCODES 136
6502 Assembler,
  error messages 281-283
slash, in Editor
  as "infinite" repeat factor 91
  as string delimiter 95, 111
slot numbers of peripherals 26, 277
slots, in SYSTEM.LIBRARY 189-193
Soroc IQ120 6, 202, 210
  SETUP procedure 212-214
SOROCGOTO.CODE 210, 211
SOROC.MISCINFO 199
string delimiters, in Editor 95,110
string replacement, in Editor 108-113
substitute string, in Editor 110
summary, Assembler 289
  Assembler directives 157-172
  Command options 20
  Compiler 289
  Editor commands 125-126
  Editor, use of 288
  Filer commands 67-69
  Filer, use of 287
  Linker 289
  operating system 285-290
  utilities 219-222, 290
SYMBOLTABLE DUMP, in Assembler 142
syntax diagram, assembly file 151
  file specification 29
  volume specification 27
SYSCOM communications,
  P-machine 226-227
system reconfiguration 199-209, 219
SYSTEM.APPLE 7, 274
SYSTEM.ASSEMBLER 136-137, 275
SYSTEM.CHARSET 274
SYSTEM.COMPILER 128-129, 274
SYSTEM.EDITOR 72, 274
SYSTEM.FILER 24, 274
SYSTEM.LIBRARY 177-179, 186-193, 274
SYSTEM.LINKER 176-177, 274
SYSTEM.MISCINFO 199-201, 274
SYSTEM.PASCAL 7, 12, 274
SYSTEM.SWAPDISK 130, 137-138
SYSTEM.SYNTAX 128-129, 274
SYSTEM.WRK.CODE , see workfile
SYSTEM.WRK.TEXT , see workfile
SYSTERM: , volume name 26, 56

T

target string, in Editor 110
terminal,
  external 6, 199-213, 219-220
text files, format of 266
text formatting, in Editor 98-101
.TEXT (versus .CODE), when to use 272
The Last Assemmbler (TLA) 136
The Program,
  Pascal example 248-249, 263-264
.TITLE , Assembler directive 171, 179
T(oken default ,
  Environment option 121
  with F(ind 93
  with R(eplace 109
tracks, on diskette 25
T(ransfer , Filer command 34-41, 68
turnkey system 8
TURTLEGRAPHICS, Intrinsic Unit 176

U

UNITs 176-180
U(pdate ,
  Editor command 78, 89, 122, 126
U(ser restart,
  outer level command 19, 20
  diskfiles needed 14
utility programs
  calculator 216-218, 222
  changing GOTOXY
    communication 210-213, 220
  formatting new
    diskettes 184-186, 218
  introduction 184
  removing linefeed
    from return 214-215, 220
  summary 218-221, 290
  system librarian 186-193, 218-219
  system
    reconfiguration 199-209, 219

V

V(erify, Editor command 122, 126
V(erify option,
  with R(eplace 109, 125

- volume 26
  - shorthand volume names 28
  - specification 26-27
- volume names and numbers,
  - chart of 26, 276
- volume-specification syntax 27, 67
- V(olumes , Filer command 55-56, 69

W

- warm boot 8, 12
- W(hat , Filer command 55, 68
- wildcards 30-33
  - and C(hange command 46-47
  - and LIBRARY utility 191
  - and L(ist directory 56-58
  - and R(emove command 48-49
  - and T(ransfer command 39-41
- .WORD , Assembler directive 160, 173
- workfile 9, 29, 77-82, 84-85
  - and Asssembler 138-140
  - and Compiler 130-132
  - and Filer commands 51-55
  - clearing 55, 77
  - saving 53-55, 79-81
  - starting 55, 78
  - updating 78
- W(rite , Editor command 123, 126

X-Y

- X(amine , Filer command 62-64, 69
- X(change , Editor command 107, 126
- X(ecute ,
  - outer level command 17-18, 20
    - diskfiles needed 14
    - also see utility programs

Z

- Z(ap , Editor command 103-104, 126
- Z(ero , Filer command 50-51, 68

/

- as "infinite" repeat factor 91
- as string delimiter 95, 111

*

- as file size
  - specification 30, 44, 137, 139
- and P(refix command 65
- as specifier for SYSTEM.LIBRARY 193
- as volume name of system disk 28
- in assembly listings 138

∧

- as C(ommand character 115

,

- as file separator 33, 45

:

- and P(refix option 65
- as volume specifier 26, 28

$

- and T(ransfer command 36, 67
- in Assembler 138
- in Compiler 131
- in Linker 181

=

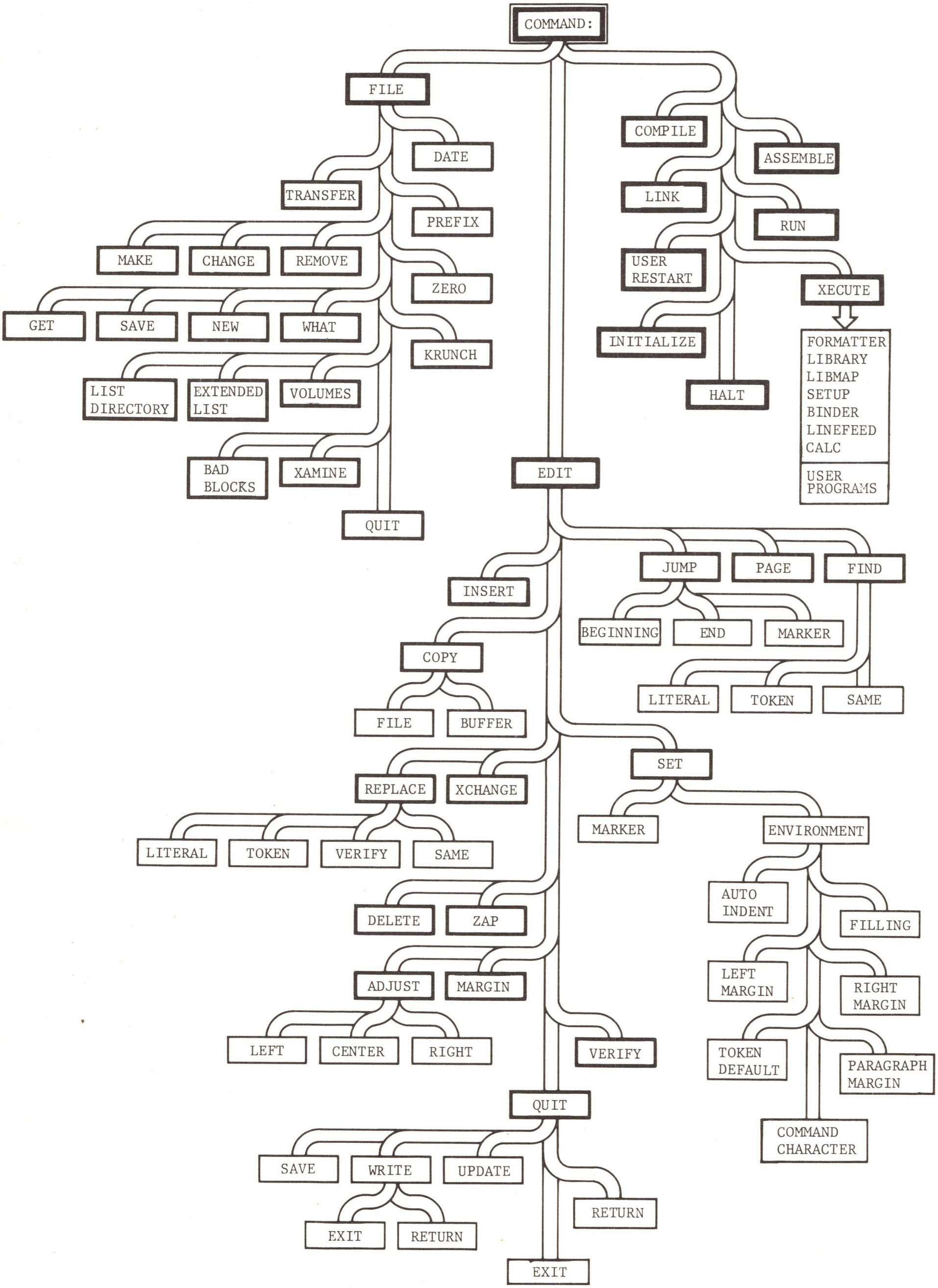
- cursor move in Editor 92, 103, 124
- wildcard in Filer 30-33
- wildcard in LIBRARY Utility 191

< and >

- to set direction in Editor 91-92,139

?

- for seeing more of prompt line 7
- wildcard in Filer 30-33
- wildcard in LIBRARY utility 191


COMMAND:
FILE
DATE
TRANSFER
PREFIX
MAKE
CHANGE
REMOVE
ZERO
GET
SAVE
NEW
WHAT
KRUNCH
LIST DIRECTORY
EXTENDED LIST
VOLUMES
BAD BLOCKS
XAMINE
QUIT
COMPILE
ASSEMBLE
LINK
RUN
USER RESTART
XECUTE
INITIALIZE
HALT
FORMATTER
LIBRARY
LIBMAP
SETUP
BINDER
LINEFEED
CALC
USER PROGRAMS
EDIT
JUMP
PAGE
FIND
INSERT
BEGINNING
END
MARKER
COPY
LITERAL
TOKEN
SAME
FILE
BUFFER
SET
REPLACE
XCHANGE
MARKER
ENVIRONMENT
LITERAL
TOKEN
VERIFY
SAME
AUTO INDENT
FILLING
DELETE
ZAP
LEFT MARGIN
RIGHT MARGIN
ADJUST
MARGIN
LEFT
CENTER
RIGHT
VERIFY
TOKEN DEFAULT
PARAGRAPH MARGIN
QUIT
COMMAND CHARACTER
SAVE
WRITE
UPDATE
RETURN
EXIT
RETURN
EXIT

apple computer inc.®

10260 Bandley Drive
Cupertino, California 95014
(408) 996-1010

030-0100-00